2012

Proving safety properties of software

Kang Gui
Iowa State University

Proving safety properties of software

by

Kang Gui

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Suraj C. Kothari, Major Professor

Srinivas Aluru

Tien Nguyen

Manimaran Govindarasu

Samik Basu

Iowa State University

Ames, Iowa

2012

DEDICATION

To my parents *Yousheng Gui* and *Jianping Chang*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis.

First and foremost, Dr. Suraj C. Kothari for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Srinivas Aluru, Dr. Tien Nguyen, Dr. Manimaran Govindarasu and Dr. Samik Basu. They asked enlightening questions and provide fruitful thought during my prelim exam. I would like to thank Jason Stanek, who assist me improving my understanding of STL. He also helped to implement algorithm for breaking cycles of cyclic graph. I would like to thank Jon Mathews from EnSoft Corp. who helped to use Atlas efficiently. He also helped to add functionality in Atlas used for this research. Also I would like to thank Ahmed Y Tamrawi who helped me implementing the program of getting event trace graph.

Additionally, I would like to thank all my friends in Ames who give me help and support not only in my research but also in my personal life during my Ph.D. study.

ABSTRACT

The use of software is pervasive in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, health-care, manufacturing, transportation, entertainment, and consumer appliances. Our safety, security, and economy are now closely linked to the reliability of software.

This research is about a technique to prove event-based safety properties of program. A safety property is defined in terms of event traces. An event trace is associated with an execution path and it is the sequence of events that execute on the path. Each event is identified with a program statement or a block of statements. Particularly, this research has been focused on one type of problem that follows one type of safety property we call matching pair (MP) property. Memory leaks, asymmetric synchronization, and several other defects are examples of violation of the matching pair property. The property involves matching between two types of events on every execution path. We present a practical method to validate the MP property for large software. The method is designed to address the challenges resulting from the cross-cutting semantics and presence of invisible control flow. The method has two phases: the macro phrase and the micro phrase. The macro analysis phase incorporates important notions of signature and matching pair graph (MPG). Signatures enable a decomposition of the problem into small independent instances for validation, each identified by a unique signature. The MPG(X) defines for each signature X, a minimal set of functions to be analyzed for validating the instance. The micro analysis phase produces the event traces graph representing all the relevant execution paths through the functions belonging to a MPG(X). A fast and accurate analysis of large software is possible because the macro analysis can exactly identify the functions that need to be analyzed and the micro analysis further greatly reduces the amount of analysis required to cover all execution paths by creating event trace graph (ETG) from the control flow graph (CFG). We applied macro level analysis on eight versions of Linux kernels spanning

for three years. We further calculated ETGs for all functions identified by macro analysis for three versions of Linux. With the combination of macro and micro analysis, we were able to prove the correctness of more than 90% of the synchronization instances in the Linux kernel. For each remaining case, we produced relevant ETGs for the further investigation by human experts.

CHAPTER 1. OVERVIEW

The use of software is pervasive in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, healthcare, manufacturing, transportation, entertainment, and consumer appliances. Our safety, security, and economy are now closely linked to the reliability of software. Software is vulnerable to unintentional programming errors and also intentional malicious attacks. In the event of a crash, a personal computer can be rebooted, but in a safety-critical system like an automobile or an airplane, a crash can mean loss of human lives and waste of millions of dollars. And it happened; in 1996, the Ariane 5, a $500 million rocket launched by the European Space Agency, exploded 40 seconds after lift-off due to a software error in the guidance system. There is a pressing need for a new technology to ensure reliability of software.

In his invited talk at the 2009 International Symposium on Software Reliability Engineering (ISSRE), Juichi Takahashi, a Distinguished SONY Engineer, pointed out the challenge of analyzing even software as ubiquitous as the Linux kernel (about one million lines of C code) for a common software problem: mismatches between locking and unlocking. Unmatched locking can lead to deadlocks in the system and can cause the system to hang and have to be rebooted. Takahashi presented a graph to show how the number of test cases grows exponentially (from 6 to 11340) as the number of concurrent threads increases from 2 to 6. Our analysis shows that a configuration of a recent Linux kernel has 1446 threads that use locking. A complete testing of software is not possible.

Static analysis has gained importance as another approach to detect software defects. Unlike testing, the static analysis covers all paths, however, it aggregates the execution results along different paths and that typically leads to many false positives. Another serious issue with static analysis is that it cannot tackle invisible control. We will use the term invisible control to refer to

changes in control flow that are not visible through static control flow analysis based on control structures or function calls. The invisible control can happen due to interrupt processing, thread scheduling, and exception handling. The runtime binding can also be thought of as another form of invisible control.

This paper is about a technique to prove event-based safety properties of program. A safety property is defined in terms of event traces. An event trace is associated with an execution path and it is the sequence of events that execute on the path. Each event is identified with a program statement or a block of statements. For example, a locking event is a statement that executes the `mutex_lock(p)` call, similarly an unlocking event that executes the `mutex_unlock(q)` call. The `p`, and `q` denote the objects to be locked or unlocked. A safety property is stated as a condition that every event trace must satisfy. For example, an event E_1 to lock an object is safe if it is succeeded by an event E_2 to unlock the same object on all execution paths containing the event E_1. A proof of this safety property involves proving if each locking event is safe or not. For a safe locking event, the proof provides all the corresponding unlocking events. For an unsafe locking event, the proof provides an execution path on which the locking event is not followed by a corresponding unlocking event. We have done a case study in which the proof technique was applied to three versions of the Linux kernel. The detailed results of our case study are posted on the Web. A summary of results and a few representative examples are included in the paper.

Clearly, a key challenge for designing such a proof technique is the exponentially large number of execution paths. A sequence on n successive `if-then-else` statements results in 2^n execution paths. Moreover, a function call causes a path to splits into many paths. Our major contribution is a set of abstractions that make it possible to examine all the event traces in spite of the large number of execution paths. The abstraction has two parts: a macro-model to tackle the explosion of execution paths through function calls and a micro-model to tackle the explosion of execution paths within each individual function. The micro-model has a broader applicability, it can be used for any event driven analysis of programs. The macro-model is defined for a property that we call the matching-pair property. This property essential requires matching events must happen. It is illustrated here for the locking and unlocking events but it is

also applicable in many other contexts, for example, memory allocation and deallocation must match or interrupt-enable must match with interrupt-disable. The macro-model is designed so that it is also possible to tackle the invisible control flow.

1.1 Dissertation Outline

The detailed outline of this dissertation is as follows:

Chapter 1 introduced the importance of the matching pair problem and event based program analysis. This chapter also describes the objectives of this dissertation.

Chapter 2 reviews the related literature for program analysis. They advantages and limitations provides us guideline to develop our event base approach.

Chapter 3 explained the matching pair problem and the limitations of current research. A 2-phase method has been introduce the overcome some of the limitations, especially for inter-procedure matching. To identify the functions related to the matching pair problem is critical to solve the problem. Reverse call graph is an approach to the solution. While it still suffer from the large number of functions involved. Matching pair graph is introduced as an refined call relation graph for the problem. Matching pair graph refines the call graph by applying a sequences of queries. The ending result is the minimum number of functions that must be analyzed for proving the matching pair property. At function level (micro level), path analysis method is introduced.

Chapter 4 presents an empirical study of matching pair property for mutex locking/unlocking in 8 versions of Linux kernels. The complex of the matching pair problem and the amount of reduction by applying the MPG are presented.

Chapter 5 presents a technique to prove event-based safety properties of program. The proving is separated into 2 parts where micro model and macro model. The abstraction has two parts: a macro-model to tackle the explosion of execution paths through function calls and a micro-model to tackle the explosion of execution paths within each individual function. The micro-model has a broader applicability, it can be used for any event driven analysis of programs. The macro-model is defined for a property that we call the matching-pair property. This property essential requires matching events must happen. It is illustrated here for the

locking and unlocking events but it is also applicable in many other contexts, for example, memory allocation and deallocation must match or interrupt-enable must match with interrupt-disable. The macro-model is designed so that it is also possible to tackle the invisible control flow.

Chapter 6 summaries the contribution of this dissertation.

CHAPTER 2. RELATED WORKS

The increasing importance of software reliability has led to a large body of research aimed at identifying violation of various program safety properties, including memory leak and mismatch of safe synchronization lock [6, 36, 25, 28, 21, 22, 35]. We restrict the related work discussion to the techniques that are most relevant to our work.

2.1 Finding Defects of Large Source Code

The challenge for large system is of course of the size of of the code base and time for the analysis. The research and tools for large system especially operating systems have been limited. Dynamic approaches generally not work because of its run time overhead, while static analysis fail to scale to the large code base. [36] proposed a static tools for detect memory access errors for large software including operating systems. Their tools generates reports for potential defects, but provide no good way to validate the results.

2.2 Graph Based Program Analysis

It has been a long time since software is represented as a graph or a collection of graph [17]. Graph representations like control flow graph (CFG), call graph and program dependency graph (PDG) are among the most commonly used ones. There are many tools that can generate call graph for inter-procedure analysis [31, 26]. Of all of the above tools, none but Atlas from EnSoft [15] has the capability to generate a subgraph of a call graph containing minimum necessary functions for analyzing the security property.

2.3 Events Based Program Analysis

Joshi et.al [25, 28] proposed a dynamic technique for finding deadlocks of concurrent Java program. They first observe certain important operations during and execution and construct a skeleton of the execution. Next they run off the shelf model checking tools on the program skeleton and report potential deadlock. Similar to our approach, they can also generate event trace to the potential defects for manual investigation. But their approach is limited in two ways. First, their approach does not guarantee the coverage of all execution paths. Second, they have to annotate the code which is time consuming and limits its application to large codes.

2.4 Function Summary

To proof safety properties of infinite-state system like software, it is critical to apply function abstraction. Techniques for abstracting software are a prerequisite to make proving techniques like software model checking applicable. The history of function summary dates back to early 1990s [20]. There have been many researches on automatic construction of function summary for model checking [1, 23, 4, 3, 18].

2.5 Other Related Works

There is a more general approach to problems of safety checking by describing the safety property as a finite state machine. Some of them use flow-sensitive data-flow analysis to track the state at each program point and detect violations in the state machine, such as Metal [13] and ESP [9]. Some others use model checking [2, 37, 10, 5]. These approaches allow users to specify arbitrary state machines to check the violation of the safety property. The complexity of state machine prevent them from scaling to larger systems.

CHAPTER 3. A 2-PHASE METHOD FOR VALIDATION OF MATCHING PAIR PROPERTY WITH CASE STUDIES OF OPERATING SYSTEMS

The reliability of software continues to be more important than ever before. If the IBM 360 malfunctions, the only result is an incorrect calculation. If an avionics system malfunctions, hundreds of people could die. And it happened. A software module for Ariane 4 was incorrectly modified for Ariane 5, which led to its explosion forty seconds after the launch [11]. Even recently, a software glitch in Toyota's control system software has resulted in a massive recall of vehicles. Software validation and verification is a critically important area where we need significant new advances. The challenge lies in inventing cost-effective methods for validation and verification of large software. This paper is about an engineering approach that combines mathematical rigor with pragmatic simplifications to solve a difficult validation problem with many practical benefits. It is an approach to provide a good combination of the mechanical processing power of tools and the intelligent decision making power of humans.

We present an analysis framework for semi-automated validation of software with respect to the *matching pair* (MP) *property*. The framework is useful for validating software against memory leaks, asymmetric synchronization, and several other defects. These critical defects can all be viewed as violations of the MP property. The framework is designed to analyze large and complex software. We demonstrate the analysis framework by presenting two case studies of validation: (a) the memory leak problem for the Xinu kernel with 5,000 lines of code, and (b) the asymmetric synchronization problem for the Linux kernel with one million lines of code.

The paper [7] states the MP property for analyzing memory leaks and introduces the notion of *guarded flow analysis*. We use the following general formulation of the MP property: on any execution path, the identifier used by a call to P must flow into and be used by exactly one

call to V, where P and V are problem-specific events in the software. This formulation can be applied to analyze different types of problems. For example, P and V can be function calls to allocate and deallocate memory, and the identifier is the pointer to the allocated memory. A memory leak happens when a memory allocation call is not followed by a memory deallocation call on at least one execution path. As another example, P can be the function call to lock mutex, V the function call to unlock mutex, and the identifier a variable of type mutex. An asymmetric synchronization is defined as an unmatched execution where a call to lock mutex is not followed by a call to unlock mutex. We will refer to the P and V calls as *matching events*. The unique identifier X associated with P (or V) will be called the *signature*. $P(X)$ and $V(X)$ will denote calls with signature X.

The rest of the paper is organized as follows. Section 2 provides an overview of the 2-phase method. Section 3 describes the macro analysis phase. Section 4 describes the micro analysis phase. Section 5 discusses the validation process. Case studies with the Xinu and the Linux kernels are presented in section 6. Section 7 gives the conclusion.

3.1 An Overview of the 2-Phase Method

A simple instance of the MP problem is one where $P(X)$ and $V(X)$ are invoked within the same function f. It is a simple instance because it involves just one function f that needs to be analyzed to validate the MP property. A complex instance of the MP problem is one that involves *cross-cutting semantics*. Instead of being invoked within the same function, $P(X)$ and $V(X)$ are invoked by different functions g and h respectively. In a complex instance, one must identify these two functions g and h and analyze at least those two functions. These functions must be analyzed together because they are connected by the use of a common signature X. The passage of the signature from g to h must be traced during the analysis. This passage may involve several functions all of which have to be analyzed to validate the particular instance of the MP problem. This cross-cutting semantics can get even more complicated. Instead of just two functions f and g, there can be multiple functions that invoke P and V using the same signature X. Moreover, the passage from f to g may not be visible through the control flow in instances where f and g are invoked by different threads. The control flow will also not be

visible if f or g is invoked by an interrupt driven routine. These cases of invisible control flow are especially difficult to validate.

In general, cross-cutting semantics makes it hard to ensure the MP property and thus increases the probability of defects [12]. The algorithmic complexity of a context-sensitive static analysis solution grows exponentially [27, 33]. Many tools [26, 8, 14, 16, 34] can perform fast intra-procedural static analysis. However, they face challenges in performing the inter-procedural analysis for analyzing cross-cutting semantics. We have identified three types of challenges that static analysis tools face in analyzing complex instances of the MP problems. We have designed the proposed 2-phase method to address these important challenges. Next, we will identify these three types of challenges.

The first challenge, resulting from cross-cutting semantics, is to identify a minimal set of functions to analyze for a given instance of the problem. First, we can form groups of functions using signatures. We can form a group $G(X)$ of functions that invoke P or V using the same signature X. But these are not the only functions to be analyzed. Suppose a function g invokes $P(X)$ and h invokes $V(X)$. It is important to identify and analyze also the functions that pass the signature X from g to h. Clearly, the reverse call graph (RCG) of $G(X)$ would suffice but it is an overkill in many cases. So, the first challenge is to define a good candidate for the minimal set of functions to be analyzed for a given signature X. We propose the notion of the matching pair graph MPG(X) as a good candidate for defining such a minimal set of functions.

The second challenge comes from the multiplicative growth of execution paths. The number of execution paths is typically small within a well designed function. However, in a complex instance, we may have a large number of functions to analyze for a signature X and the number of inter-procedural execution paths may become very large. In conservative static analysis techniques, this explosion of execution paths is managed by aggregating the results of analysis along different branches of a control structure. However, the accuracy is partly lost and the resulting analysis leads to false negatives. For example, we can get false negatives in cases where $V(X)$ is invoked on some but not all branches of a control structure So, the second challenge is to find an execution path analysis method that is efficient and also accurate. To deal with this problem, we propose path analysis (PA) tables with event-based aggregation of

paths as a part of our 2-phase method. We do not aggregate execution paths where the events are different. At the same time, we propose several optimization to minimize the explosion of execution paths to be analyzed.

The third challenge comes from the invisible control. Ordinarily, the control flow is visible through control structures and call sequences. As discussed earlier, complex instances of the MP problem may involve invisible control flow. In cases involving invisible control, static analysis tools report false positives because their analysis cannot see the connection between these invocations of $P(X)$ and $V(X)$ by different threads. Thus, the third challenge is to design a validation method that works correctly in presence of invisible control. The proposed notion of MPG(X) captures all the necessary functions that need to be analyzed for a signature X even in cases of invisible control flow. For example, the MPG(X) will include functions g and h that invoke $P(X)$ or $V(X)$ even if g and h belong to different threads.

3.1.1 Two Phases

Macro Analysis: The purpose is to identify all independent instances of the matching pair problem, and determine the minimal set of functions to be examined for each instance. The macro analysis phase has two steps:

1. **Compute Signatures:** Determine all the independent instances of the matching pair problem by computing signatures. There is one signature for each instance. In section 3, we define the notion of signature and illustrate it with a code example.

2. **Compute Matching Pair Graphs:** For each signature X, compute the matching pair graph, MPG(X). The MPG(X) is defined and illustrated in section 3.

Micro Analysis: The micro analysis is done using the proposed *Path Analysis* (PA) method that produces tables called PA tables to summarize functions. A set of PA tables are constructed, one for each function belonging to MPG(X). Each instance is validated using this set of PA tables. The micro analysis phase is discussed in detail in section 4.

This 2-phase method decomposes the validation problem into multiple instances. Each instance is identified by a unique signature X. The method minimizes the number of functions

to be examined for each instance by computing the MPG(X) for that instance. We will illustrate how the micro analysis also becomes more efficient because of the MPG(X).

We have automated the macro analysis by writing a program using the queries supported by the Atlas tool from EnSoft[1]. The initial compilation and indexing in Atlas takes about half an hour for the Linux kernel. After the initial compilation, the macro analysis including the computation of all signatures and the MPG(X) for each signature gets done in less than a minute on a PC. The path analysis method for the micro analysis is partly automated. The control paths are determined using the Understand C/C++ tool and then PA tables are written manually.

Later, we will present case studies to show how the 2-phase method provides a fairly accurate approach to validate the MP property. We will show examples to illustrate how some of the false positives and false negatives, typically reported by static analysis, are avoided in the 2-phase method.

In addition to being efficient and accurate, the 2-phase method has other benefits. First, it can be easily speeded up by employing a team of people. This is possible because the micro analysis is done on separate instances of validation. Second, the 2-phase method produces PA tables as structured artifacts. Besides validation, these PA tables can serve as valuable documentation for program comprehension. Third, the method allows incremental validation. For a new version of the software, we need to validate only the new or the modified instances of the matching pair problem. New instances can be identified by identifying the new signatures and the modifications can be detected by comparing the MPG(X) for the old and the new version.

3.2 Macro Analysis Framework

As discussed in the earlier overview, the macro analysis phase introduces signatures to separate instances on the MP problem and it introduces MPG(X) as a minimal set of functions to be analyzed to validate the instance identified by a signature X. This section provides details of signatures, the MPG(X) and how it is computed.

[1] http://www.ensoftcorp.com

3.2.1 Signatures for Matching Pair Instances

Let D (stands for direct callers) be the set of functions that invoke either P or V, or both. The first task is to divide the set D into separate groups of functions corresponding to different instances of validation. A subset of functions in D are grouped together because the calls to P and V they make are related. Consider the matching pair problem for memory leaks where the P is an allocation of memory and the V is a deallocation of memory. In this case, a group includes functions that invokes P to allocate memory M and functions that invoke V to deallocate the same memory M. For example, on one execution path f_1 allocates M to create an object X, on a separate but related execution path f_2 allocates M to create the same object X, and subsequently a function g, on a common branch belonging to the two execution paths, deallocates M. These functions f_1, f_2, and g are grouped together.

We propose the notion of signature to form groups of functions in D. We define a signature to be a global variable or a user-defined type associated with the P or V calls.

First, we will explain the case of a global variable as a signature. In this case, P and V use a global variable X as a parameter. The Linux kernel has several `mutex_lock(`X`)` and `mutex_unlock(`X`)` calls where X is a global variable. For the asymmetric synchronization problem, the `mutex_lock(`X`)` and `mutex_unlock(`X`)` are the P and V functions respectively.

Next, we will explain the case of a user-defined structure as a signature. To understand this case, consider the following example of a cross-cutting matching in Xinu kernel. In this example the `getbuf()` (memory allocation) must match with `freebuf()` (memory deallocation). The function `dswrite()` calls `getbuf()`, the pointer p to the allocated memory is passed as a parameter to the function `dskenq()` which in turn passes it as a parameter to the function `dskqopt()`. Both `dskenq()` and `dskqopt()` assign a pointer to a global structure `dsblk`. `dsinter()`, an interrupt-driven function, gets the pointer to the allocated memory from `dsblk` and deallocates the memory by invoking `freebuf()`. We will discuss the validation for this example later.

In the above example, the memory is allocated for a user-defined structure type `dreq`. The pointer that gets passed to different functions is always a pointer to the type `dreq`. `dsinter()`

also uses a pointer to the type `dreq` when it deallocates memory by calling `freebuf()`. In this example, it makes sense to use the user defined type `dreq` as the signature and group together the functions that call `getbuf()` or `freebuf()` using a pointer to `dreq`.

In a properly designed software, either a global variable or a pointer to a user-defined type is used with the P and V calls. This practice enhances the readability of the program. Without this practice, it is very difficult for a programmer to keep track of cross-cutting cases of matching and ensure the matching is correct. In our method, the cases where this practice is not followed are reported as violations of a good design practice. We will report the violations of this practice in the Linux kernel we have examined.

3.2.2 Matching Pair Graph

We will define the *matching pair graph* for signature X and denote it by MPG(X). It is intended to serve as the smallest graph that includes all call sequences of functions that need to be examined for checking the instance of the matching pair problem with signature X. We will give an algorithm to compute the MPG(X).

Let $P(X)$ and $V(X)$ denote the matching pair calls with signature X. Let RCG(X) be the reverse call graph with $P(X)$ and $V(X)$ as leaves. Clearly, MPG(X) must be a subgraph of RCG(X). Let us first discuss some examples to motivate the formal definition of MPG(X).

Figure 4.1 shows 3 examples of RCG(X). In figure 4.1(a), function `A2` calls function `A1` which calls both $P(X)$ and $V(X)$. The matching sequence of calls are contained in `A1` and there is no need to include `A2` in MPG(X).

But the function `B3` in figure 4.1(b) is a different story. It indirectly calls $P(X)$ and $V(X)$ via `B1` and `B2` respectively. In this case, it is essential to include all the nodes in MPG(X) to check the matching pair property.

Figure 4.1(c) is a more complicated case. Since `C1`, `C2`, and `C3` invokes $P(X)$ and $V(X)$, they must be included in MPG(X). The node `C4` has a path via `C1` to invoke $P(X)$ and its matching with $V(X)$ can only be checked by including `C4`. Another possibility is that `C4` is called by `C7` and the matching happens via `C7` and `C3`. Since `C7` has a path via `C3` to invoke $V(X)$ and its matching with $P(X)$ can only be checked by including `C7`. Thus `C7` must be

included independent of its relationship to `C4`. The nodes `C5` and `C6`, in a similar situation as the node `A2` in figure 4.1(a), do not need to be included in MPG(X).

3.2.3 Formal Definition of MPG(X)

Let RCG(X) be the reverse call graph with $P(X)$ and $V(X)$ as leaves. MPG(X) is an induced subgraph of RCG(X) defined as follows. Let n and m be nodes in the graph $RCG(X)$. n is *adjacent* to m iff $n = m$ or $\exists$ a sequence S of function calls in $RCG(X)$, where S starts at n and ends at m. Let $A(x, y)$ be the predicate "x is adjacent to y". A node n is *balanced* if $A(n, P(X)) \wedge A(n, V(X))$ is true, *unbalanced* if $A(n, P(X)) \wedge A(n, V(X))$ is false. Next, we define the *unbalanced child* (UBC) property. A node n has the property UBC iff n has an unbalanced child c. Now, we will define MPG(X) using the UBC property. MPG(X) is the largest induced subgraph of RCG(X) with the constraint that all of its roots have the property UBC.

Note that a node with the UBC property is not necessarily a root of MPG(X). For example, in Figure 4.1(c), `C4` has the property UBC but it is not a root.

3.2.4 Computing MPG(X)

We have implemented an algorithm for computing MPG(X) as a sequence of Atlas queries. A powerful feature of Atlas is that the queries are composable. This makes it possible to write a compact program for computing MPG(X). This query-based program is shown Figure 4.3. The queries used in the program work as follows. Let N be set of all functions. The query `Call(F)` returns set $G = \{g \in N |\ g$ calls f, where $f \in F\}$. The query `CG(F)` returns the set $G = \{g \in N |\ f$ is adjacent to g, where $g \in F\}$. The query `RCG(F)` returns the set $G = \{g \in N |\ g$ is adjacent to f, where $f \in F\}$. In addition to queries, we use the set operations supported by the Atlas query language.

The query-based algorithm finds all the nodes with the UBC property. Then, UBC nodes are used to generate the MPG(X). To illustrate the algorithm, we simulate it on the graph shown in Figure 4.1(c). The simulation results of each step are shown in Figure 3.2.

```
RCG-P = RCG(P(X))
RCG-V = RCG(V(X))
RCG-B = RCG-P ⋂ RCG-V
RCG-C = RCG-P ⋃ RCG-V
RCG-P-ONLY = RCG-P - RCG-B
RCG-V-ONLY = RCG-V - RCG-B
C-P-ONLY = Call(RCG-P-ONLY)
C-V-ONLY = Call(RCG-V-ONLY)
MPG-BAL = (C-P-ONLY ⋃ C-V-ONLY) ⋂ RCG-B
UBC = MPG-BAL ⋃ RCG-P-ONLY ⋃ RCG-V-ONLY
MPG = CG(UBC) ⋂ RCG-C
```

Figure 3.1 Atlas queries for calculating MPG(X)

```
RCG-P = {P(X),C1,C2,C4,C5,C6,C7}
RCG-V = {V(X),C2,C3,C4,C5,C6,C7}
RCG-B = {C2,C4,C5,C6,C7}
RCG-C = {P(X),V(X),C1,C2,C3,C4,C5,C6,C7}
RCG-P-ONLY = {P(X),C1}
RCG-V-ONLY = {V(X),C3}
C-P-ONLY = {C1,C2,C4}
C-V-ONLY = {C2,C3,C7}
MPG-BAL = {C2,C4,C7}
UBC = {P(X),V(X),C1,C2,C3,C4,C7}
MPG = {P(X),V(X),C1,C2,C3,C4,C7}
```

Figure 3.2 Each step of query results for the example shown in Figure 4.1(c)

3.3 Micro Analysis Framework

The macro analysis determines MPG(X) which is the set of functions to be analyzed for validating the MP property for signature X. The purpose of the micro analysis is to perform the detailed analysis necessary for validating the MP property. During the micro analysis, execution paths are analyzed for each function that belongs to the MPG(X). We use the concept of events [29] to minimize the number of execution paths that need to be examined. We call this the Path Analysis (PA) method. The results of the PA method are noted in a tabular form using a notation similar to the tabular notation introduced by Parnas et. al. for the Trace Function Method (TFM) [30]. We refer to these results as PA tables. Finally, we will discuss how the validation can be done using PA tables.

3.3.1 Event-Based Path Optimization

Developers use top-down, bottom-up or mixed approach to create a mental picture of the code. When using a bottom-up approach, developers start off by looking for interesting events manifested in the code and gradually abstract out the details. The events of interest vary depending on the developer's concern. For example, when a developer is trying to isolate memory leaks in the code, the events of interest would be memory allocations and deallocations, aliasing of pointers, and pointer escapes through function calls. The tool CVision [29] was designed to enable users to isolate and navigate through the code based on such events. The CVision is based on the notion of *event view* defined as the minimal subgraph of a call graph or reverse call graph that is relevant to a specified set of events. The MPG(X) can be thought of as an event view for analyzing the matching pair problem. Here, we will discuss an event-based optimization designed as a part of the PA method.

A set of events is defined for a given type of the MP problem. Each event is associated with code statements which execute the particular event. We will refer to these as *marked code* statements. For validating the matching pair property, we consider the following types of events: *invocations* of functions belonging to the MPG(X), the return statements to exit a function, the *escape events*, and the *capture events*. The escape events are the events through

which the signature escapes out of a function. The escape events are: (a) the signature is returned by a function g, (b) the signature is passed as a parameter to a function f, (c) the signature is assigned to an external entity, typically a global variable. The capture events are: (a) the signature returned by a function g is captured in a function f, (b) the signature that came in as a read by a function f, (c) the signature is read from an external entity, typically a global variable.

Figure 3.3 shows an example of a function with relevant events marked. This function is part of the validation example we will discuss later. Here, we will use this example to illustrate the event-based optimization technique to reduce the number of relevant execution paths that need to considered for the purpose of validation. First, we will discuss the relevant events and markings. The signature is `dreq` and so the deallocation call `freebuf(drptr)` is marked. The call `freebuf(drptr->drbuff)`is not marked because the signature is not `dreq`. The `return` statements are marked to note that the execution paths end.

The event-based path optimization works as follows. The optimization principle is: treat a set of paths as one equivalent path if the sequence of events is the same on those paths. This makes sense because the validation logic would be the same for those paths. The optimization principle implies that the set of paths due to control structures with no marked events on any of its branches can be treated as one equivalent execution path. This reduction is illustrated in Figure 3.3. As shown in the figure, `dsinter()` has seventeen execution paths which reduce to three by using event base optimization. Moreover, the two paths with only the `return` are further combined into one path also by the event optimization. Thus the original seventeen execution path are reduced to two.

Note that the MPG(X) does help in making the event-based path optimization more effective. Referring back to Figure 3.3, the `dsinter()` code has a call to the function `dskstrt()`. This call is not marked because `dskstrt()` does not belong to MPG(`dreq`) which makes that call a non-relevant event. If that call were to be treated a relevant event, the path reduction would not be as effective, it would have resulted in four paths instead of two.

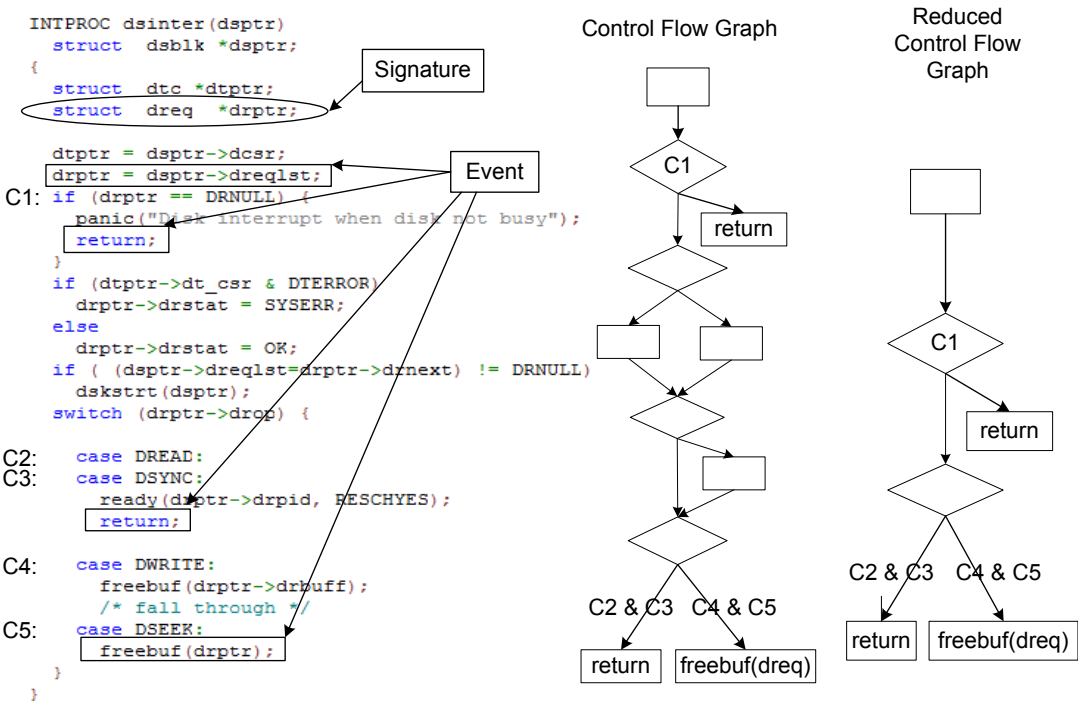


Figure 3.3 `dsinter()` and its reduced control flow graph

3.3.2 Path Analysis Method

For a given function, the Path Analysis (PA) Method, as suggested by its name, analyzes and represents the execution paths of a function. Using the event-based optimization discussed earlier, the PA method minimizes the number of execution paths that need to be analyzed. The PA method results in a summarization of the function. The summarization is presented using the PA table, a tabular notation we have designed.

The PA table is defined for each function g belonging to the MPG(X). It has the following structure. The columns of the PA table are broken into three sections: Input, Condition, and Event Trace. The header row of the PA table marks these three sections. The Input section lists the external entities available to the function g. These entities are either global variables or parameters passed to g by its caller. The Condition section will have as many columns as the conditional branch statements in the function after applying the event-based path optimization. The header row shows the condition labeled as $C_i(f)$ for a function f. The Event Trace section describes the sequence of events on an execution path of the function.

Note that an execution path shown in the PA table is an equivalent execution path obtained by applying the event-based path optimization to actual execution paths.

Each row in a PA table corresponds to one equivalent execution path. As we go across a row, the entry in each Condition column is T(TRUE), F(FALSE), or − (*a non-affecting condition*) on a particular execution path. The − entry may be there because the path does not hit this condition because of an exit by a previous `return`. Another reason for having the − entry is to represent the `fall through` logic of the `CASE Statement` in the C language. The PA tables for `dsinter()` is shown in Table 3.5.

3.4 Validation Using the 2-Phase Method

The 2-Phase method decomposes the matching pair problem into separate instances which can be validated independently. For each instance, we produce the signature X, the MPG(X) and the PA tables for functions belonging to the MPG(X). The validation is done using these PA tables. We will describe the overall process of validation and then illustrate it with an example from the Xinu kernel.

3.4.1 The Validation Process Using PA Tables

First, a few important observations about PA tables. Each row of a PA table represents a set of equivalent execution paths through the function f. The sequence of events on each execution path are listed in the last column. If the sequence includes a call to a function g, then the execution path continues and it can have multiple branches through g. An execution path ends in f itself if the sequence of events does not include a function call.

Suppose f calls g and g calls h, and the execution path ends in h. Then a *complete execution path* is obtained by concatenating the paths, with one path selected from each of the functions f, g, and h. The sequence of events on the complete execution path is obtained by concatenating the sequences of events on the selected paths.

The validation process works as follows:

1. For each invocation of $P(X)$, start with the PA table for the function f and the row for

the execution path where the invocation event is shown.

2. Iterate over all complete execution paths obtained by following the call chains as described above.

3. Check the sequence of events on the complete execution path to validate if $P(X)$ is followed by a unique $V(X)$. If not, it is a violation of the MP property.

3.4.2 Important Optimizations for Validation

The number of complete execution paths can grow multiplicatively along a call chain. Suppose g has three paths, h has four paths, and we have call chain where f calls g and g calls h, then a single path in f can branch into twelve paths. This number can become very large quickly with long call chains.

The multiplicative growth can be effectively restricted in practice by applying the following optimizations:

1. **MPG(X)**: The MPG(X) eliminates unnecessary call chains by minimizing the number of functions to be examined. Also, as illustrated earlier, the MPG(X) makes the next optimization more effective.

2. **Event-Based Path Optimization**: It minimizes the number of paths within a function by retaining only the paths where events relevant to the matching pair problem happen.

3. **Condition-Based Path Optimization**: It minimizes the number of complete paths by eliminating certain path formation based on incompatibility of the governing conditions for individual path segments selected from various functions.

The first two are the new optimizations reported in this paper. The third optimization has been used in various contexts [38]. We will quickly explain the optimization in the context of the PA method. The governing condition for an execution path in a function is easily obtained from the PA table as a logical `AND` of the conditions along the row that corresponds to the path. Continuing the above example, suppose $G(f)$ be the governing condition for the path

in f, and $G_1(g)$, $G_2(g)$, $G_3(g)$ be the governing conditions for three paths in g, and $G_1(h)$, $G_2(h)$, $G_3(h)$, $G_4(h)$ be the governing conditions for four paths in h. It is possible to rule out several paths by checking the compatibility of the governing conditions. For example, it may be the case that if $G(f)$ holds then $G_2(g)$, $G_3(g)$, $G_4(h)$ cannot be true, and if $G_1(g)$ holds then $G_2(h)$, $G_3(h)$ cannot be true. Then, instead of twelve complete paths, there would be only one complete path corresponding to the allowable combination $G(f)$, $G_1(g)$, $G_1(h)$.

3.5 Case Study Results

We present two case studies of validation using the 2-phase method. The first study is for validating the Xinu kernel with respect to the MP property applied to check the memory leak problem. The second study is for validating the Linux kernel with respect to the MP property applied to check the asymmetric synchronization problem.

3.5.1 Xinu Case Study

Xinu is a small multi-threaded operating system used as a teaching tool in academia and as a small kernel for embedded systems in industry. Its relatively small size, about 5,000 lines of code, makes it a good choice for illustrating concrete applications of the 2-phase method.

Management of buffer pools is a critical functionality in Xinu. The system buffers are allocated by invoking the function `getbuf()` and deallocated by invoking the function `freebuf()`. To apply the 2-phase method, P is mapped to `getbuf()` and V is mapped to `freebuf()`. When `getbuf()` is invoked, a global variable representing the buffer pool id is passed as an argument. The `getbuf()` returns the pointer to the allocated memory which is cast as a pointer to some user-defined type. In example

```
1 drptr = (struct dreq *) getbuf(dskrbp);
```

the pointer is cast to the type `struct dreq`. The global buffer pool id and the user-defined type are used as signatures in this case study. These are the artifacts that help the developer to track and distinguish different instances of buffer allocations.

Table 3.1 Signatures in Xinu

Type	`getbuf()` calls	`freebuf()` calls	MPG(X)
dreq	4	8	9
epacket	8	12	31

Table 3.2 PA table for `dswrite()`

Input	Event Trace
`devsw`	`getbuf(dreq)` `dskenq(dreq, devsw.dvioblk)`

The Xinu kernel we analyzed has 263 functions. Our macro analysis using the Atlas tool resulted in two signatures `dreq` and `epacket`. The results including the MPG(X) sizes are shown in Table 3.1.

3.5.2 An Example of Validation

We will present an example of validation for the signature `dreq`. The MPG(`dreq`) is shown in Figure 3.4. In this example, we will check if the `getbuf()` call in `dswrite()` has a matching `freebuf()` call on every execution path.

Function `dswrite()` does not call `freebuf()` so this is an example of the cross-cutting matching. `dswrite()` does have an escape event where the pointer to the allocated memory is passed as a parameter to `dskenq()`. The PA table for `dswrite()` is shown in Table 3.2.

The PA table for `dswrite()` shown in Table 3.2 has one row indicating that `dswrite()` has only one execution path. This execution path must continue because its event trace shows the invocation of `dskenq()`. Next, we examine the PA table for `dskenq()` shown in Table 3.3. This PA table shows four execution paths out of which three terminate and one continues with

Table 3.3 PA table for `dskenq()`

Input	Condition			Event Trace
	C1	C2	C3	
`dreq, devsw.dvioblk`	T	-	-	`devsw.dvioblk.dreqlst` ← `dreq`
	F	T	-	`dskopt(devsw.dvioblk.dreqlst, dreq)`
	F	F	T	`devsw.dvioblk.dreqlst` ← `dreq`
	F	F	F	`devsw.dvioblk.dreqlst` ← `dreq`

a call to `dskqopt()`. Here, it looks like a memory leak because these three execution paths that terminate at `dskenq()` have no `freebuf()` call. We will do a closer examination of this potential leak later. For now, we follow the path into `dskqopt()`.

Instead of referring to actual conditions in the code, we represent them symbolically in PA tables. For example, in Table 3.3, we have used `C1`, `C2`, and `C3`. The actual conditions can be rather long and complex. Also, each row corresponds to one equivalent path which can correspond to several actual paths through code with different conditions and we will need to represent them all. The actual conditions are not important in this analysis unless it is the case where we apply condition based optimization that was discussed earlier.

Function `dskqopt()` is a little more complicated, it contains seven paths as shown in Table 3.4. Thus, we have potentially ten complete execution paths starting from `dswrite()`, of which three terminate at `dskenq()`. Similar to the three paths that terminated at `dskenq()`, the PA table for `dskqopt()` shows four paths which have no `freebuf()`. Again, this looks like a memory leak.

The number of execution paths shown in a PA table can be further reduced by applying the condition-based path optimization discussed in the previous section. Let us apply that optimization. `dswrite()` sets the flag `drop==DWRITE`, and makes the statement `dsptr->dreqlst == DRNULL` false. This information is pass to `dskenq()` and `dskqopt()`. Based on these two conditions, only two paths in the PA table remain feasible. One path contains the `freebuf()`, and the other one contains an escape event but no `freebuf()`.

So far we have five complete execution paths of which three terminate in `dskenq()` and two terminate in `dskqopt()`. Before we proceed further we want to point out the opportunity for a more powerful event-based path optimization. Of these five complete execution paths, four have the same event trace containing one and the same escape event `devsw.dvioblk.dreqlst` ← `dreq`. Thus, we have to examine really only two execution scenarios corresponding to the two execution paths. In one execution scenario we do have a matching and so it is not a problem. The other execution scenario appears to be a memory leak and we will examine it more closely.

This last scenario brings us to a point where automated analysis is not enough but provides

Table 3.4 PA table for `dskqopt()`

Input	Condition						Event Trace
	C1	C2	C3	C4	C5	C6	
`dreq` `devsw.dvioblk.dreqlst`	T	-	-	-	-	-	
	F	T	-	-	-	-	`freebuf(dreq)`
	F	F	T	-	-	-	`freebuf(dreq)`
	F	F	F	T	-	-	`freebuf(dreq)`
	F	F	F	F	T	-	
	F	F	F	F	F	T	`devsw.dvioblk.dreqlst` ← `dreq`
	F	F	F	F	F	F	

Table 3.5 PA table for `dsinter()`

Input	Condition					Event Trace
	C1	C2	C3	C4	C5	
`dsptr` `dsptr` ← `devsw.dvioblk`	T	-	-	-	-	`drptr` ← `dsptr->dreqlst;`
	F	T	-	-	-	`drptr` ← `dsptr->dreqlst;`
	F	F	T	-	-	`drptr` ← `dsptr->dreqlst;`
	F	F	F	T	-	`drptr` ← `dsptr->dreqlst;` `freebuf(dreq)`
	F	F	F	F	T	`drptr` ← `dsptr->dreqlst;` `freebuf(dreq)`

very valuable evidence for human experts to complete the validation task. One important piece of evidence − all execution paths that have appearance of a memory leak do have an escape event. Just as a side note, it is not very likely that the developer would forget memory deallocation on several paths and instead place a mysterious escape event there. We will now examine other important evidence from the 2-phase method that provides valuable insight to complete the validation.

First, the MPG(`dreq`) shown in Figure 3.4 has the node `dsinter()` that calls only `freebuf()`. This `freebuf()` probably matches with some `getbuf()`, which one? Let us now check the PA table for `dsinter()` shown in Table 3.5. One path in `dsinter()` is feasible under the condition `drop==DWRITE` set by `dswrite()`. The event trace on this path, shows a capture event followed by `freebuf(dreq)`. The paths with suspected memory leaks have escape events and here it is a matching capture event. In functions `dskenq()` and `dskqopt()` the signature `dreq` escapes to the global data structure `devsw.dvioblk.dreqlst` and function `dsinter()` captures it from the same data structure. This is strong evidence for a match.

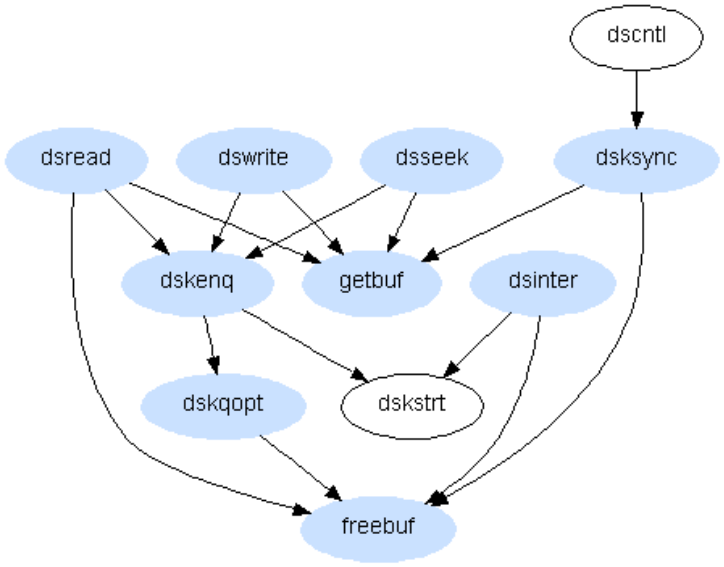


Figure 3.4 Reverse call graph related to signature `dreq`. Shadowed nodes belong to MPG(X)

Table 3.6 Reductions from the event-based path optimization (EPO) and condition-based path optimization (CPO)

Function	Execution Paths	After EPO	After CPO
`dswrite`	1	1	1
`dsread`	1	1	1
`dsseek`	1	1	1
`dsksync`	1	1	1
`dskenq`	4	4	1
`dskqopt`	7	7	1
`dsinter`	17	2	1

Function `dswrite()` is the most complicated case for the signature `dreq`. The other cases are not discussed here in detail but the results of path optimizations are reported in Table 3.6.

3.5.3 Mutex Synchronization in Linux

The results of macro analysis for the Linux kernel version 2.6.31 are reported here. The kernel has 1,081,090 lines of code and 39,973 functions, we found that the validation task can be broken down into 249 instances of synchronization, with the average of eight functions to be examined per instance. The distribution of the size of MPG(X) is presented in Table 3.7. It can be seen that the size of MPG(X) is small in majority of the cases. The size is less than

Table 3.7 Distribution of the size of MPG(X)

Range	Distribution				Average size
	≤ 5	$6 \rightarrow 10$	$11 \rightarrow 50$	> 50	
Count	186	35	25	3	8.24

five in 186 out of 249 instances of the MP problems. The size is bigger than 50 in only three instances.

The savings from MPG(X) are significant. The size of MPG(X) is typically much smaller than that of the reverse call graph RCG(X). An illustration of MPG(X) for one signature in Linux is shown in Figure 3.5. This figure shows the reverse call graph for the signature X = `super_block`. The RCG(X) has 52 functions of which 24 belong to the MPG(X). The functions belonging to the MPG(X) are highlighted in Figure 3.5. Of the functions belonging to the MPG(X), all but one - `fsync_super()`, satisfy the UBC property and eleven of those functions are the roots of the MPG(X).

We did a study with fifteen graduate students, where each student did the micro analysis of the same five instances of synchronization, with the size of MPG(X) varying from five to fifty functions. On average, students reported ten hours for validating all five instances. Since the size of the MPG(X) is small in many instances as shown in Table 3.7, we estimate that after a few hours of training with the 2-phase method, a good software engineer can complete the validation of the entire Linux kernel in roughly 150 hours.

3.6 Conclusion

The paper presents a 2-phase method for validating the MP property for large software. The important innovations are: the use of signatures to decompose the validation problem into independent instances each associated with a unique signature X, the notion of MPG(X) as a minimal set of functions to be analyzed for the instance with signature X, and the PA method for analyzing the execution paths efficiently and accurately.

We present a case study of the XINU kernel to show how the 2-phase method provides a fairly accurate approach to validate the MP property. We illustrate how some of the false

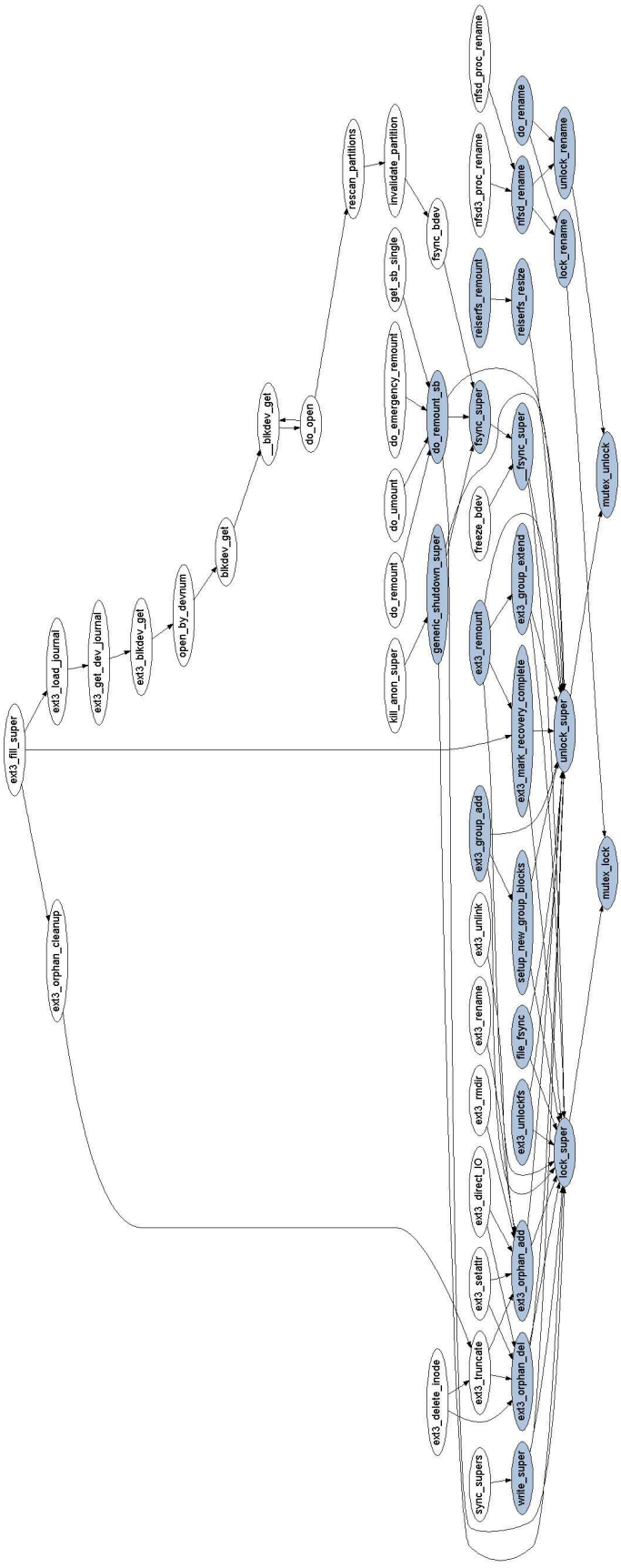


Figure 3.5 Reverse call graph of signature `super_block` with MPG(X) highlighted

positives and false negatives, typically reported by static analysis, are avoided in the 2-phase method. Specifically, we identify three challenges for static analysis and illustrate how those challenges are addressed by the 2-phase method. This Xinu case study illustrates a difficult case of validation that simply cannot be done using static analysis because it involves an interrupt-driven routine and thus the control flow is not visible for performing static analysis.

We present another case study which is about the Linux kernel. The results of macro analysis for the Linux kernel version 2.6.31 are reported in the paper. The kernel has 1,081,090 lines of code and 39,973 functions. We found that the validation task can be broken down into 249 instances of synchronization. The average size of MPG(X) is eight, i.e., eight functions to be examined per instance. This knowledge of number of instances and the size of each instance provides a basis for estimating the effort needed for validation.

We have proposed MPG(X) as a minimal set of functions to be analyzed for an instance of the MP problem with signature X. The Linux case study includes an example to show how the MPG(X) can be much smaller compared to the reverse call graph which one can use as an alternative to MPG(X). We present the distribution of the size of MPG(X). The size of MPG(X) is less than five in 186 out of 249 instances. The size is bigger than 50 in only three instances.

This 2-phase method is amenable to automation. Currently, the macro analysis phase is completely automated using the Atlas tool from EnSoft. This automation is done by writing program based on Atlas queries. The paper presents an algorithm to compute MPG(X) as a sequence of Atlas queries. The micro analysis phase, i.e., the PA method is currently semi-automatic with scope for further automation.

The paper presents a semi-automated approach where the analysis can handle simple cases automatically and for complex cases it provides valuable evidence for a human expert to complete the validation. The paper presents an example where it would be difficult for a human expert to reason without the evidence produced by the automated analysis based on the 2-phase method.

CHAPTER 4. PATTERN BASED EMPIRICAL STUDY TO ASSIST WITH ANALYSIS OF MATCHING PAIR PROPERTY

There is a pressing need for new technologies to ensure reliability of software. A personal computer can be rebooted, but in a safety-critical system like an automobile or an airplane, a crash can mean loss of human lives and waste of millions of dollars. And it happens; in 1996, the Ariane 5, a $500 million rocket launched by the European Space Agency, exploded 40 seconds after lift-off due to a software error in the guidance system [11].

Juichi Takahashi, a Distinguished SONY Engineer, recently pointed out the challenge of analyzing even software as ubiquitous as the Linux kernel (about one million lines of C code) for a common software error: mismatches between locking and unlocking [32]. Takahashi presented a graph to show how the number of test cases grows exponentially - from 6 to 11340 as the number of concurrent threads increases from 2 to 6. Since the test cases grow exponentially, complete testing is impossible for large programs. Static analysis has been proposed as another approach [8, 14, 16, 34]. However, a complete and sound static analysis of large programs is not feasible in practice.

We propose a pattern-based method for checking the MP property. Good programmers regulate the use of P and V operations to make it easier for them to ensure the matching of MP property. To the extent possible, they design programs so that the corresponding P and V operations are performed within the same function. However, situations such as the 2-level design of device drivers require that the P is performed by a function f belonging to the upper level driver and the V is performed by another function g belonging to the lower level driver. We present two patterns that can deal with the situations where the corresponding P and V operations are performed by different functions. We present an empirical study of the Linux kernel to discover these patterns. We use the mutex synchronization in Linux as the example

where the P and V operations are mapped to functions `mutex_lock()` and `mutex_unlock()` respectively.

4.1 Identifier Pattern

There are different instances of matching based on the mutex lock used by P and V operations. An identifier X for the lock is passed as a parameter to P and V operations. The identifier pattern is a simple method a programmer could use to keep track of the identifier. Without it, tracking the identifier would require pointer alias analysis which can become very complicated [24].

It is simple to keep track of the identifier if the programmer uses either a global variable or a user defined type as the identifier. The global variable or the user-defined type become a unique token to track the identifer and we call it the *Identifier Pattern*. We expect extensive use of this pattern in any well designed software. As an example, in one version of the Linux kernel we analyzed, 1181 out of 1220 instances of P or V operations use the *Identifer Pattern*.

4.2 Matching Pair Graph Pattern

Identifier Pattern disassembles the matching pair problem into several small pieces which could be analyzed separately. But it doesn't reduce the total number of functions to be checked in the matching pair analysis. In a well designed software, the set of functions involved in the matching for each identifier X will be much smaller than the set of functions in the reverse call graph (RCG) of the identifer X, denoted by RCG(X). We propose another pattern which we call *Matching Pair Graph (MPG) Pattern*. We propose the notion of MPG(X), typically a much smaller subset of RCG(X) as the set of functions involved in the matching for each identifier X.

4.2.1 Matching Pair Graph

We will define MPG(X) as the smallest graph that needs to be examined for checking the matching pair property for signature X. The bigger the MPG(X), the more work it will be to

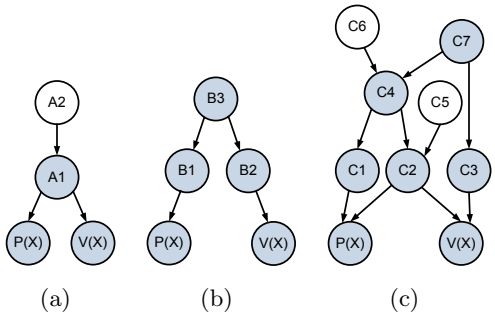


Figure 4.1 Examples of RCG(X)- shadowed nodes belong to MPG(X)

check the matching pair property. Thus, the desired pattern is that MPG(X) be small.

Let X be the lock associated with a synchronization concern. Let $L(X)$ and $U(X)$ be the locking and unlocking operations on X. Let RCG(X) be the reverse call graph with $L(X)$ and $U(X)$ as the leaves. We are interested in defining the matching pair graph MPG(X) of lock X as the smallest graph that includes all call sequences of functions that need to be examined for checking the matching pair property for the lock X. Clearly, MPG(X) must be an induced subgraph of RCG(X).

Let us first discuss some examples that we can use to motivate MPG(X).

Figure 4.1 shows 3 examples of RCG(X). In figure 4.1(a), function `A2` calls function `A1` which calls both $L(X)$ and $U(X)$. The matching sequence of calls are contained in `A1` and there is no need to include `A2` in MPG(X).

But `B3` in figure 4.1(b) is a different story. Though `B3` doesn't call `L(X)` or `U(X)` directly, it links the locking and unlocking sequences via `B1` and `B2` respectively. It is essential to include all the nodes in MPG(X) to check matching pair property.

Figure 4.1(c) is a more complicated case. Since `C1`, `C2`, and `C3` perform the locking/unlocking operations they must be included in MPG(X). The node `C4` has a path via `C1` to invoke `L(X)` and its matching with `U(X)` can only be checked by including `C4`. Another possibility is that `C4` is called by `C7` and the matching happens via `C7` and `C3`. Since `C7` has a path via `C3` to invoke `U(X)` and its matching with `L(X)` can only be checked by including `C7`. Thus `C7` must be included independent of its relationship to `C4`. The nodes `C5` and `C6`, similar situation as the node `A2` in figure 4.1(a), do not need to be included in MPG(X).

4.2.2 Definition of MPG(X)

Let RCG(X) be the reverse call graph with $P(X)$ and $V(X)$ as the leaves. MPG(X) is an induced subgraph of RCG(X) defined as follows. Let n and m be nodes in the graph $RCG(X)$. n is *adjacent* to m iff $n = m$ or $\exists$ a sequence S of function calls in $RCG(X)$, where S starts at n and ends at m. Let $A(x, y)$ be the predicate "x is adjacent to y". A node n is *balanced* if $A(n, P(X)) \wedge A(n, V(X))$ is true, *unbalanced* if $A(n, P(X)) \wedge A(n, V(X))$ is false. Next, we define the *unbalanced child* (UBC) property. A node n has the property UBC iff n has an unbalanced child c. Now, we will define MPG(X) using the UBC property. MPG(X) is the largest induced subgraph of RCG(X) with the constraint that all of its roots have the property UBC.

Be aware that MPG(X), instead of the formal validation technique, is a pattern we observed from the real code for the MP problems. It does not always guarantee to give a precise boundary for the matching pair problem if the behavior of the function related to P and V operations depends on its input. One such pseudo example is presented in Figure 4.2. Design a function in which the behavior depends on other functions isn't good. But even for examples like this, the MPG(X) stills provides the lower bound for the analysis. What's more, from our analysis of Xinu and Linux code, we haven't be aware of any violation of this kind.

```
void g(bool b){if(b) lock(); else unlock()};
void f(){g(true);g(false)}
```

Figure 4.2 MPG pattern fail on this case

4.2.3 Computing MPG(X)

We implemented the algorithm for computing MPG(X) as a sequence of Atlas queries [15]. The Atlas tool supports a query language. A powerful feature of Atlas is that the queries are composable which made it possible for us to write a compact program for computing MPG(X). We have shown this query-based program in Figure 4.3. This algorithm is straightforward. Starting from L(X) and U(X), for each function, the algorithm checks its adjacency to L(X)

and U(X). Then the algorithm marks the functions that satisfy the UBC property. At the end, MPG(X) is computed as the maximum induced subgraph of RCG(X) using node with UBC property as the roots.

```
RCG-P = RCG(P(X))
RCG-V = RCG(V(X))
RCG-B = RCG-P ⋂ RCG-V
RCG-C = RCG-P ⋃ RCG-V
RCG-P-ONLY = RCG-P - RCG-B
RCG-V-ONLY = RCG-V - RCG-B
C-P-ONLY = Call(RCG-P-ONLY)
C-V-ONLY = Call(RCG-V-ONLY)
MPG-BAL = (C-P-ONLY ⋃ C-V-ONLY) ⋂ RCG-B
UBC = MPG-BAL ⋃ RCG-P-ONLY ⋃ RCG-V-ONLY
MPG = CG(UBC) ⋂ RCG-C
```

Figure 4.3 Atlas queries for calculating MPG(X)

Clearly, the reverse call graph RCG(X) includes all call sequences of functions that need to be examined for checking the matching pair property. The advantage of MPG(X) is that it can be much smaller than RCG(X) - it means less work for checking the matching pair property. For the `super_block` lock, MPG(X) is 54% smaller compared to RCG(X). As a justification for using MPG(X) instead of RCG(X), we have presented later empirical results for the Linux kernel.

4.3 Empirical Study Setup

We presented two patterns, *Identifier Pattern* and *MPG Pattern*, which we believe to be useful for checking the matching pair property. To validate the usefulness of these patterns, we conducted an empirical study over 9 versions of Linux kernels spanning from year 2006 to 2009. In this section, we describe the experimental setup and other details in this empirical study.

4.3.1 Experimental Setup

The Linux kernel is a good candidate for the empirical study - it is open source, wildly used, and less likely to be biased by individual design pattern used by a small portion of developers.

The kernel itself is highly dynamic and critically depends on synchronization mechanisms[1]. Mutex-based synchronization mechanisms are heavily used in Linux kernel.

Table 4.1 give an overview of the 9 versions of the Linux kernel, including release date[2], source lines of code (SLOC), number of functions and files. The Atlas tool uses the gcc compiler to compile and build the code. For this study, the code was compiled for the x86 architecture.

Version	Release Date	SLOC	Files	Functions
2.6.16	20-Mar-06	559,870	1939	18486
2.6.18	20-Sep-06	532,651	1845	16294
2.6.20	4-Feb-07	701,580	2147	21576
2.6.22	8-Jul-07	696,574	2169	21991
2.6.24	24-Jan-08	758,861	2362	24382
2.6.26	13-Jul-08	797,448	2452	26075
2.6.28	24-Dec-08	978,071	3173	35113
2.6.30	10-Jun-09	1,043,399	3190	38192
2.6.31	9-Sep-09	1,081,090	3400	39973

Table 4.1 Summary of 9 versions of Linux kernels

4.3.2 Identification of Lock Operations

In Linux, the functions `mutex_lock(`X`)` and `mutex_unlock(`X`)` are lock and unlock operations, where X is the lock identifier. After the introduction of the functions `mutex_lock()` and `mutex_unlock()` in the Linux version 2.6.16, the subsequent versions use predominantly these new functions and rarely the other locking functions `mutex_trylock()`, `mutex_lock_interruptible()` and `mutex_lock_killable()`. In our analysis, we do not make a distinction between these different types of locking functions.

4.4 Experimental Results

The experimental study is conducted to address the following 2 questions: 1) In how many places is the *Identifier Pattern* used in Linux? 2) Does the *MPG Pattern* provides small result in Linux? We will answer these two questions in this section. In this section, we use $|\text{MPG}(X)|$ to denote the size of the $\text{MPG}(X)$.

[1] http://www.ibm.com/developerworks/linux/library/l-linux-synchronization.html
[2] Dates as shown at http://www.kernel.org

Version	Calls `lock()` or `unlock()`	Lock Identifiers			Per Lock operations
		Globals	User Types	Locals	
2.6.16	173	8	6	0	12.40
2.6.18	434	41	36	5	5.69
2.6.20	425	64	35	15	4.35
2.6.22	436	67	40	15	4.11
2.6.24	522	86	49	22	3.91
2.6.26	554	83	54	33	3.96
2.6.28	952	127	84	39	4.49
2.6.30	1123	140	96	39	4.77
2.6.31	1220	148	101	39	4.92

Table 4.2 Identifer pattern usage in Linux kernels

4.4.1 Identifier Pattern Usage

We will start with an overview of synchronization in Linux kernel. The amount of synchronization usage increases as the Linux involves. The number of functions invoke the lock or unlock operations increase from 173 to 1220. So does the usage of the Identifier Pattern as shown in Table 4.2. The number of global identifer increases from 8 to 148, while the user type identifer increases from 6 to 101. In all 9 versions of Linux kernels, 5715 out of the entire set of 5922, which is about 97%, P or V operations are associated with either global variable identifers or the user defined type identifiers for locks. In the remaining 3% cases, the identifiers are local variables and P and V operations are performed in the same function. These results show that the identifier pattern is used heavily across all 9 versions of Linux kernels we have tested. The average operations per identifer keeps stable excepts the initial version in which the mutex synchronization is firstly used in Linux. This can be interpreted as independent matching groups are added without affecting the existing groups. In other sense, separate the locking operations based on our identifier pattern is effective.

4.4.2 MPG Pattern Size

The distribution of the size of MPG(X) is presented in Table 4.3. The average size of MPG(X) size is around 8, which is small, except the initial version. Besides, we would also

like to know if MPG(X) is small enough for majority of the cases. From Table 4.3, we see that for majority of the locks, the size of MPG(X) is less than the average 8. For example, for 75% (186 out of 249) of the locks the |MPG(X)| is less than the average for the latest version 2.6.31. In no more than 3 signatures do the size of MPG(X) be bigger than 50 which we think is large.

Version	|MPG(X)| Range				Average |MPG(X)|
	≤ 5	$6 \rightarrow 10$	$11 \rightarrow 50$	> 50	
2.6.16	8	2	3	1	16.29
2.6.18	54	13	9	1	8.79
2.6.20	80	12	5	2	7.87
2.6.22	90	13	3	2	7.49
2.6.24	112	16	5	2	7.57
2.6.26	112	19	5	1	6.97
2.6.28	163	29	16	3	7.70
2.6.30	183	29	21	3	8.06
2.6.31	186	35	25	3	8.24

Table 4.3 Distribution of the |MPG(X)|

An import property about MPG(X) is that it should provide signature reduction compared with the size of the RCG(X). We presents the amount of reduction data graph from RCG(X) to MPG(X) in Table 4.4. The average reduction is about 50% while the maximum reduction for one signature can be as high as 99%. This data gives the empirical evidence that using MPG(X) instead of RCG(X) in the analysis does simplify the analysis.

Version	Percentage of Reduction	
	Average	Maximum
2.6.16	56%	95.86%
2.6.18	50%	98.55%
2.6.20	50%	99.18%
2.6.22	49%	99.20%
2.6.24	48%	99.52%
2.6.26	47%	99.20%
2.6.28	51%	99.60%
2.6.30	53%	99.89%
2.6.31	56%	99.82%

Table 4.4 Reduction from RCG(X) to MPG(X)

4.5 Conclusion and Future Works

The paper presents an empirical study of the mutex synchronization in 9 versions of Linux kernels. It checks for two patterns that a programmer is likely to use as a good design which makes it easier to check the MP property - a necessary condition for ensuring correct synchronization. The first pattern, called the *identifier pattern*, states that a programmer is likely to use a global variable or a user defined type for the mutex lock. This pattern can greatly simplify validation of the MP property because it provides a unique identifier to track a lock without performing a complex data flow analysis. The unique identifier is called the *signature of the lock*. The second pattern is based on the *matching pair graph*, the MPG(X), as the smallest graph that needs to be examined for checking the matching pair property where X is the signature for a lock. The bigger the MPG(X), the more work it is to check the matching pair property. Thus, the desired pattern is that the MPG(X) be small. The paper presents an automated method to compute the MPG(X) using the Atlas tool from EnSoft.

From the results of the empirical study we can see that the two patterns are followed extensively in the Linux kernel. Of the totality of signatures from the nine versions, 97% follow the identifier pattern. The average size of the MPG(X) is 8 with only three signatures where the size is bigger than 50.

These patterns split the validation work into multiple pieces, each one of them identified by the unique signatures for a lock. For each piece, MPG(X) provides the set of functions that need to be examined in detail by taking into account the control flow within the functions. The detailed analysis may reveal that the MPG(X) is not sufficient and more functions are need to examine the matching for the particular lock. However, the case where the MPG(X) does not suffice are likely to be small in number, and indicative of either bad design or additional patterns that need to be captured. One such example, indicative of bad design, is presented in the paper.

CHAPTER 5. PROVING MATCHING PAIR PROPERTY - A CASE STUDY WITH LINUX KERNEL

5.1 Challenges of Matching Pair Property

We define a safety property, called the Matching Pair (MP) property as a unified way to analyze a class of defects in which events must happen in matching pairs. For example, a memory leak is a violation of the MP property where a memory allocation is not matched with memory deallocation. In this paper, the MP property is discussed with mutex locking and unlocking as the matching events.

Let's denote L as the locking event, U as the unlocking event. o represent object to be locked/unlocked. Given an execution path P, a locking event L_o is *safe on* P if it is followed by U_o on P, with no L_o in between. Locking event L_o is *safe* if and only if it is safe on all execution paths.

Definition 1 *Matching Pair (MP) Property: Given software satisfies the MP property if and only if every locking event in the software is safe. A locking event that is not safe will be called a violation of the MP property.*

One challenge is the exponential growth of the number of execution sequences. The number of possible execution paths grows exponentially with the number of non-nested control structures. Only 3 non-nested control statements can reach 8 execution paths as shown in Figure 5.1. Functions with multiple control statements are very common. Many functions has more than 30 control statement in Linux. For example function `tc_ctl_tfilter` has 36 control statements.

Let us explain with a concrete illustration. The Figure 5.2(a) shows a control flow graph (CFG) of a function, where L_o and U_o represent locking and unlocking of an object o, c_n

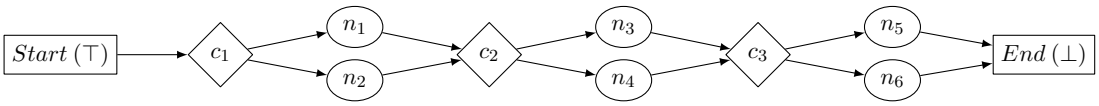


Figure 5.1 3 non-nested control statements results 8 execution paths

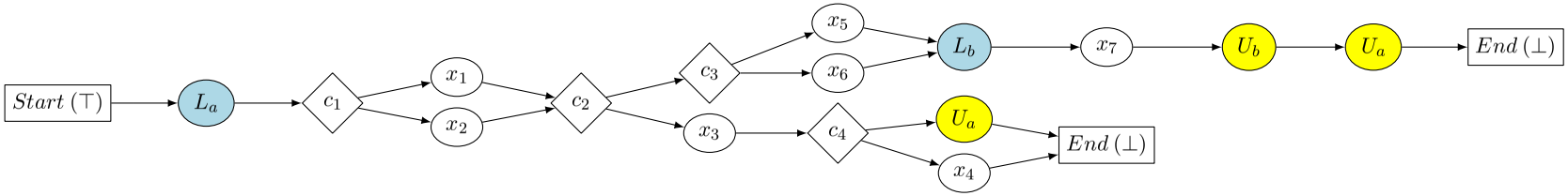
represents a condition that causes a 2-way branch, and x_m represents other events not relevant to the MP property. *Start* $(\top)$ and *End* $(\bot)$ represents the starting and ending of an execution sequence.

In this example, there are four branching conditions and eight execution sequences from the start $(\top)$ of the control flow graph to the end $(\bot)$. There are also multiple L and U events under different branches. By analyzing all eight execution sequences individually by vary the conditions of each of the four branching conditions, it is possible to prove/disapporve the MP property for this function. But there are better ways. First, note that only a subset of the branching conditions may be important for checking the MP property. For example, the branches due to the condition c_1 are not relevant because the events on both the branches do not affect the MP property. The branches due to the condition c_3 are also not relevant. The important conditions that relevant to the MP property are called *governing conditions*. The value of these conditions will decides the existence and order of important event under different execution sequences. Also we can omit irrelevant events. After doing that, what remains are three new type of event sequences called *event traces*:

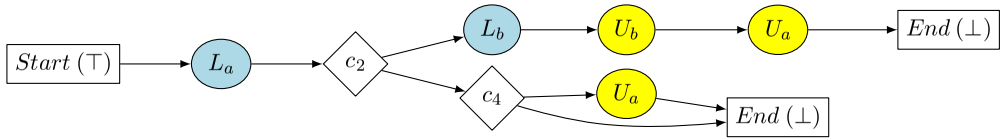
- $\top L_a \bot$, associated with 2 execution sequences;
- $\top L_a U_a \bot$, associated with 2 execution sequences;
- $\top L_a L_b U_b U_a \bot$, associate with 4 execution sequences.

Formal definition of notations and approaches to get event traces will be discussed in later sections.

The execution sequences, event traces, and the governing conditions are summarized in Figure 5.2(c). Note that the locking event L_a is not safe on two out of eight execution paths. The locking event L_b is safe; it is safe on all four execution paths on which it occurs.



(a) Control flow graph, locking and unlocking events are highlighted



(b) Refined flow graph with only locking and unlocking events and governing conditions

Execution Sequences	**Event Traces**	**Governing Conditions**
$\top L_a c_1 x_1 c_2 x_3 c_4 x_4 \bot$	$\top L_a \bot$	c_2, c_4
$\top L_a c_1 x_2 c_2 x_3 c_4 x_4 \bot$		
$\top L_a c_1 x_1 c_2 x_3 c_4 U_a \bot$	$\top L_a U_a \bot$	c_2, c_4
$\top L_a c_1 x_2 c_2 x_3 c_4 U_a \bot$		
$\top L_a c_1 x_1 c_2 c_3 x_5 L_b x_7 U_b U_a \bot$	$\top L_a L_b U_b U_a \bot$	c_2
$\top L_a c_1 x_1 c_2 c_3 x_6 L_b x_7 U_b U_a \bot$		
$\top L_a c_1 x_2 c_2 c_3 x_5 L_b x_7 U_b U_a \bot$		
$\top L_a c_1 x_2 c_2 c_3 x_6 L_b x_7 U_b U_a \bot$		

(c) Event traces, execution sequences and the governing conditions. Differences in each execution paths are highlighted

Figure 5.2 Example of execution sequences and Event Traces

It is not rare that the matching unlocking event U is not in the same function that contains the locking event L event. The inter-procedural matching, not only further increase the number of execution paths to be analyzed, but it also creates additional challenges.

The inter-procedural matching can be divided into three cases:

- **Inter-procedural matching through calls:** A function F invokes L event, there is a sequence of function calls from F to G, and the function G invokes the matching U event. See Figure 5.3(a) .

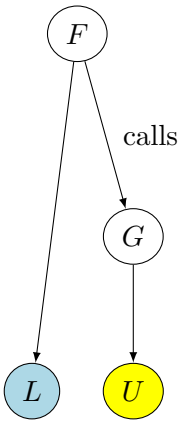
- **Inter-procedural matching through callbacks:** A function F invokes L, there is a sequence of function callbacks from F to G, and the function G invokes the matching U. G may invoke U directly or again it could be through a call sequence. The important distinction is that it involves callbacks. See Figure 5.3(b).

- **Inter-procedural matching through invisible control:** A function F invokes L, another function G invokes L, and it is not possible to reach G from F through either calls or callbacks. See Figure 5.3(c).

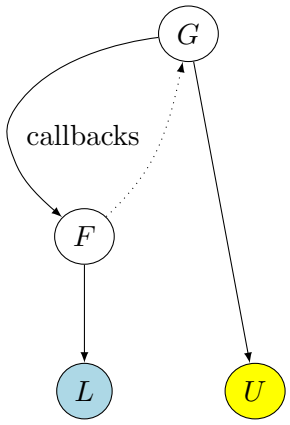
Following all the callbacks to the root nodes implies processing the reverse call graph of the functions that invoke either the L or U or both the events. The reverse call graphs have several thousand functions. It is not just about the number of functions, at every call site of each one of these functions, the number of execution paths will continue continue to explode.

There are also other issues. One is, how to tackle the locking and unlocking events that occur inside a loop. When loop exists, counting the number of execution sequences becomes a problem. Another issue is, how to handle unstructured code.

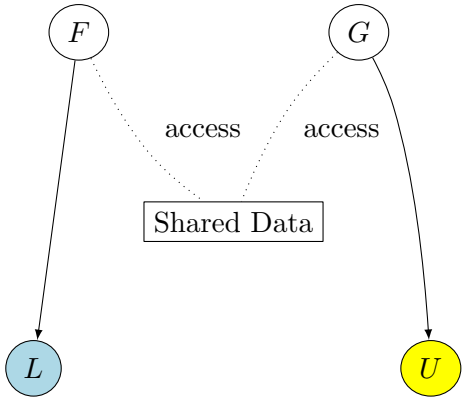
Note that the matching L and U must be for the same object. Another challenge is how to track the object associated with the locking and unlocking events. Figure 5.4 shows a code example from Linux. There are multiple L (`mutex_lock`) and U (`mutex_unlock`) operations in this code. In any execution sequence, there are at least two L events. But these L events possibly operate on two different objects `audit_cmd_mutex` and `audit_filter_mutex`. Tracking the objects is a dataflow analysis problem. In tracking the objects, one may encounter the full



(a) F connects with G through call relation



(b) F connects with G through callback relation



(c) There is no visible control flow between F and G. They may both access some shared data though

Figure 5.3 Different calling relations of inter-procedure matching

```
1 void audit_kill_trees(struct list_head *list) {
2   mutex_lock(&audit_cmd_mutex);
3   mutex_lock(&audit_filter_mutex);
4   while (...) {
5     ...
6     mutex_unlock(&audit_filter_mutex);
7     ...
8     mutex_lock(&audit_filter_mutex); }
9   mutex_unlock(&audit_cmd_mutex);
10 }
```

Figure 5.4 Code Example - Multiple locking events associate with different objects

blown complexity of the data problem or it may be possible to design an effective notion of signature to differentiate objects.

5.2 Micro Model

As seen from the last illustration, it is the event traces, i.e., the sequences of locking and unlocking events and their governing conditions, that we need to focus on for proving the MP property. The number of event traces can be much smaller than the number of execution paths. In the illustration, we had eight execution paths but only three sequences of events. The purpose of the Micro Model is to provide an efficient mechanism to get all possible event traces without having to traverse all the execution paths. We will discuss in this section the Micro Model an abstraction of the control flow graph.

The Micro Model is defined with respect to a given set S of events. An event is a discrete activity in the program. In our case study, the set S consists of locking L and unlocking U events. An event is associated with a program statement or a block of consecutive program statements. Each L and U event is associated with one particular program statement or a block of statement for a given problem. Typically, an event is executed for an object and it is important to track that object. For example, each locking or unlocking event is for a unique object. In case of mutex locking and unlocking in Linux, a pointer `p` to the object is passed as a parameter.

To properly explain the Micro Model, a few notions are defined as following:

Definition 2 *A program* execution sequence *is a sequence of program statements executed for*

a single run of a program.

Definition 3 *Given a set S of events, an* event trace *is a subsequence of a program execution sequence that includes only the events from the set S.*

We will discuss separately how the loops will be handled.

5.2.1 Event Flow Graph

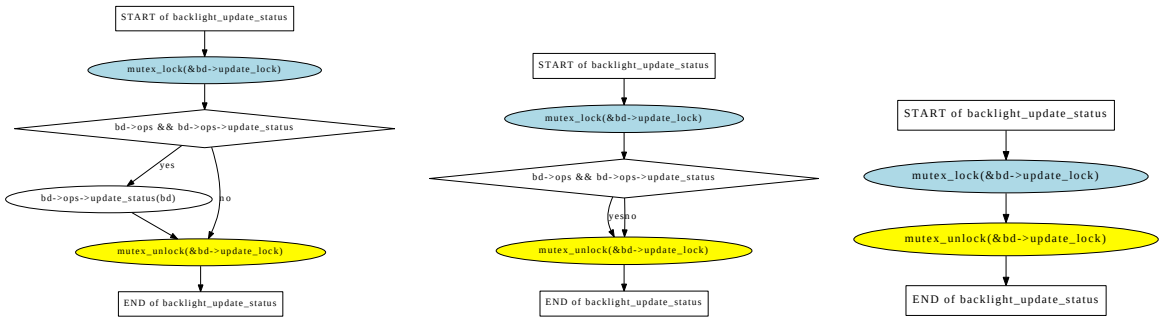
We start with the *Control Flow Graph* (CFG) for a function. We introduce two pseudo nodes: the *top node* ($\top$) where the function execution begins and the *bottom node* ($\bot$) where the function execution ends. The one or more *return* nodes connected to the $\bot$ node. The CFG is a directed graph with potential loop inside. Each execution path corresponds to a path in the CFG that begins at the top node $\top$ and ends at the bottom node $\bot$. Each program execution sequence corresponds to a unique execution path in CFG and it is the the sequence of nodes on that path. To better understand the structure of the CFG as well as illustrating the refinement process. Different nodes in the CFG, as well as other graphs defined later, are shown with different shapes. Start ($\top$) and bot ($\bot$) are shown as a box. Control branching nodes are shown as a diamond. Other statement nodes use eclipse.

The next step is to color the CFG. The *colored CFG* (CCFG) is the CFG where all the nodes corresponding to the events in the given set S are colored. For our case study, all the nodes corresponding to the locking and unlocking events are colored. The Figure 5.6(a) shows the CCFG of the function `backlight_update_status` for which code is listed in Figure 5.5. Nodes for L (`mutex_lock`) and U (`mutex_unlock`) on object of `bd->update_lock` are filled with different color.

As we can see, the size of CCFG are strictly related to the complexity of the function. The number of statements decides the number of node in the CCFG. Quite often, the CCFG is much noise for the MP property. Many nodes, edges are just irrelevant to the MP property. Refinement will make it better to solve the problem. We will explain the refine process using an example shown in Figure 5.7.

```
1 static inline void backlight_update_status(struct backlight_device *bd) {
2   mutex_lock(&bd->update_lock);
3   if (bd->ops && bd->ops->update_status)
4     bd->ops->update_status(bd);
5   mutex_unlock(&bd->update_lock);
6 }
```

Figure 5.5 `mutex_lock()` and `mutex_unlock()` are locking and unlocking operation in mutex synchronization problem with one signature



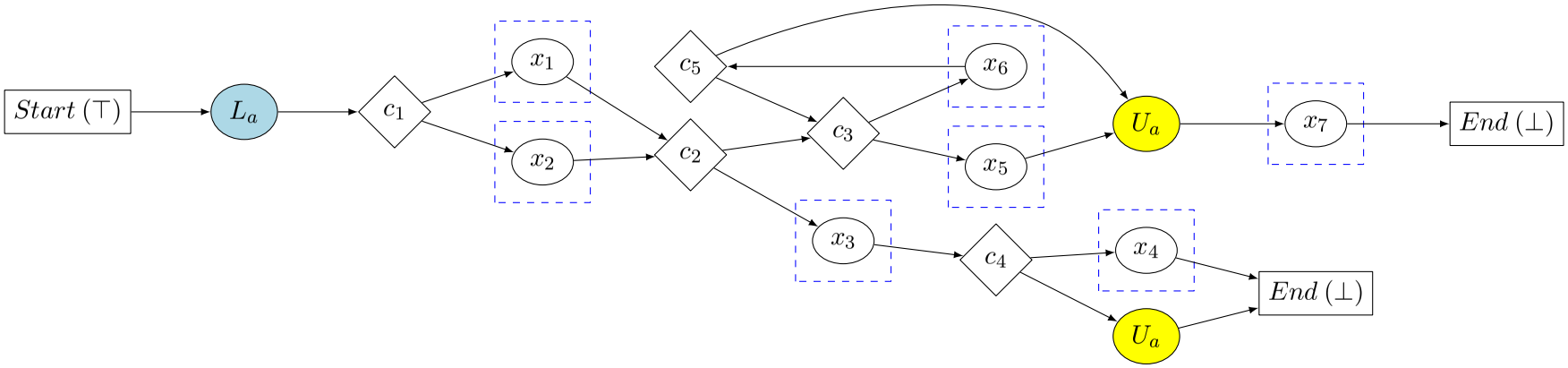
(a) CCFG (b) EFG (c) ETG

Figure 5.6 Graph representations of function shown in Figure 5.5)

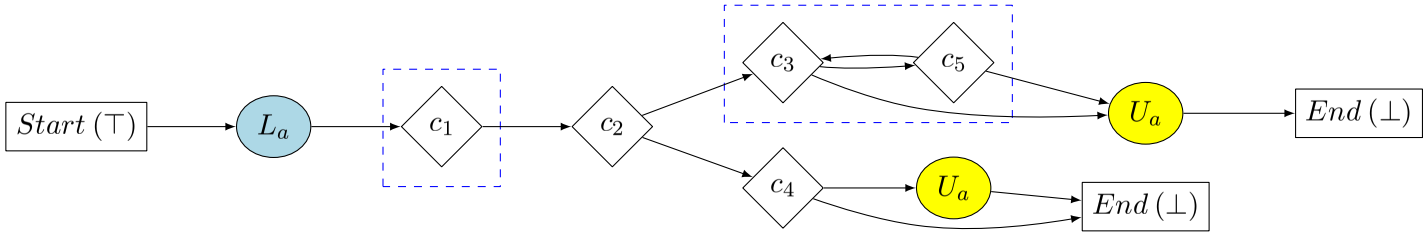
Definition 4 Event Flow Graph (EFG) *is the CCFG in which only the colored and the condition nodes are retained.*

EFG refine the the CCFG for the first step. Compared with CCFG, none relevant nodes are eliminated in the EFG. Suppose node x_m is neither a highlighted event node, nor a control branching node, in CCFG. It has potentially several incoming edges but only one outgoing ones. Let (x_m, n_{out}) be the outgoing edge, (n_{in}, x_m) be any incoming edge to x_m. $\forall(x_{in}, x_m) \in$ CFG, replace it with (x_{in}, x_{out}). x_m is also removed in EFG. Note that after the reduction, after the reduction, there may exits more than one edges between two nodes in the ETG, merge these edges into one. Apply these processes multiple times until no x nodes remains.

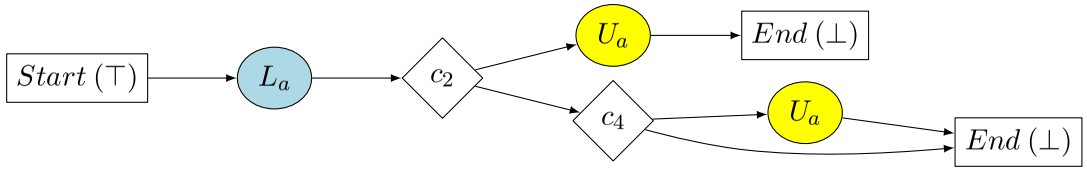
Figure 5.7(a) and 5.7(b) illustrate the refinement process. Node x_1 through x_7 are eliminated in the EFG. Edges are fixed according to the rule shown earlier.



(a) Colored CFG, nodes to be eliminated in EFG are shown in a dashed box



(b) EFG, subgraphs(with control statement) to be eliminated in ETG are shown in a dashed box



(c) ETG

Figure 5.7 Graph refine illustration from CFG to EFG to ETG

5.2.2 Event Trace Graph

Next we will define the *Event Trace Graph (ETG)* as a reduction of the EFG. We will use ETG as the Micro Model of a function. The reduction from EFG to ETG will eliminate some of the control nodes. The goal is to retain only the minimal number of control nodes for capturing the necessary event traces. No every control statement is important to the MP property. This was illustrated earlier through an example shown in Figure 5.2, now we will formalize the reduction.

Let $G = \{g_1, g_2, ..., g_n\}$ be an EFG. $B = \{b_1, b_2, ..., b_m\}$ be an induced subgraph of G with at lease two nodes. s is the set of events.

Definition 5 *Subgraph B has a* unique sink*, denoted as b_{sink},* iff

- *for any node $v \in B$, there exists a path from v to b_{sink}, and*
- *if (b_j, g_k) is an edge in G, where $b_j \in B$, $g_k \notin B$, then $b_j = b_{sink}$.*

Suppose an EFG (E) contains a subgraph B with unique sink b_{sink}. If b_{sink} is an event node or a control branching node, and other nodes in B are control branching nodes. The EFG can be transferred into a new flow graph $\bar{\text{E}}$, where subgraph B in $\bar{\text{E}}$ is replaced with node b_{sink}. Any edge $(g_k, b_j)|b_j \in B, g_k \notin B$ is replaced with (g_k, b_{sink}) in $\bar{\text{E}}$. Figure 5.7(b) highlighted the control statements to be eliminated from EFG to ETG.

Definition 6 *ETG is an event graph without subgraph with a unique sink.*

Figure 5.7(c) shows the final ETG.

Definition 7 *For any Control Flow Graph G and a set of event s, the corresponding ETG is unique.*

It's not hard to prove that for any ETG, the corresponding ETG is unique. ETG only keeps the minimum necessary control with respect to the event. Non-event statements as well as non-critical control statements are eliminated.

Definition 8 *Control branching nodes in ETG are called* governing conditions*, as they are relevant to the MP property.*

The reduction from CCFG to ETG w.r.p to number of nodes may be significant. Figure 5.8 and Figure 5.9 show the comparison between CCFG and ETG of function `acpi_device_register`. The number of nodes and control statements reduces from 51 and 10 in CCFG to only 8 and 2 in the corresponding ETG.

Each execution path is mapped to a unique path in the ETG. A large number of execution paths are mapped to a single path in the model. In our case study of the Linux kernel, the model has led to dramatic reduction of complexity in going from the CFG to ETG of a function. To illustrate this reduction, we give a concrete example of function from the Linux kernel in which 193 execution paths are mapped to 3 paths in the model. As will become clear later, the model retains all the critical information necessary for proving the safety property.

5.3 Macro Model

The Micro Model is for abstracting a function to minimize the work needed to go through execution paths within a function. The Micro Model is constructed in the form of ETG which would typically much smaller graph than the CFG. Thus, the Micro Model is a minimization of the CFG. We will now introduce the Macro Model as a minimization of the reverse call graph.

As we have discussed before, the matching events can be in two different functions and we have to construct event traces that span across functions. We defined three categories of intra-procedural mapping and noted that the analysis extends to the entire reverse call graph (RCG). Here we will discuss an abstraction, which we call the *Matching Pair Graph* (MPG, which typically is much smaller than the RCG and it suffices to prove the matching pair property.

As shown in Figure , function calls are represented with shape hexagon.

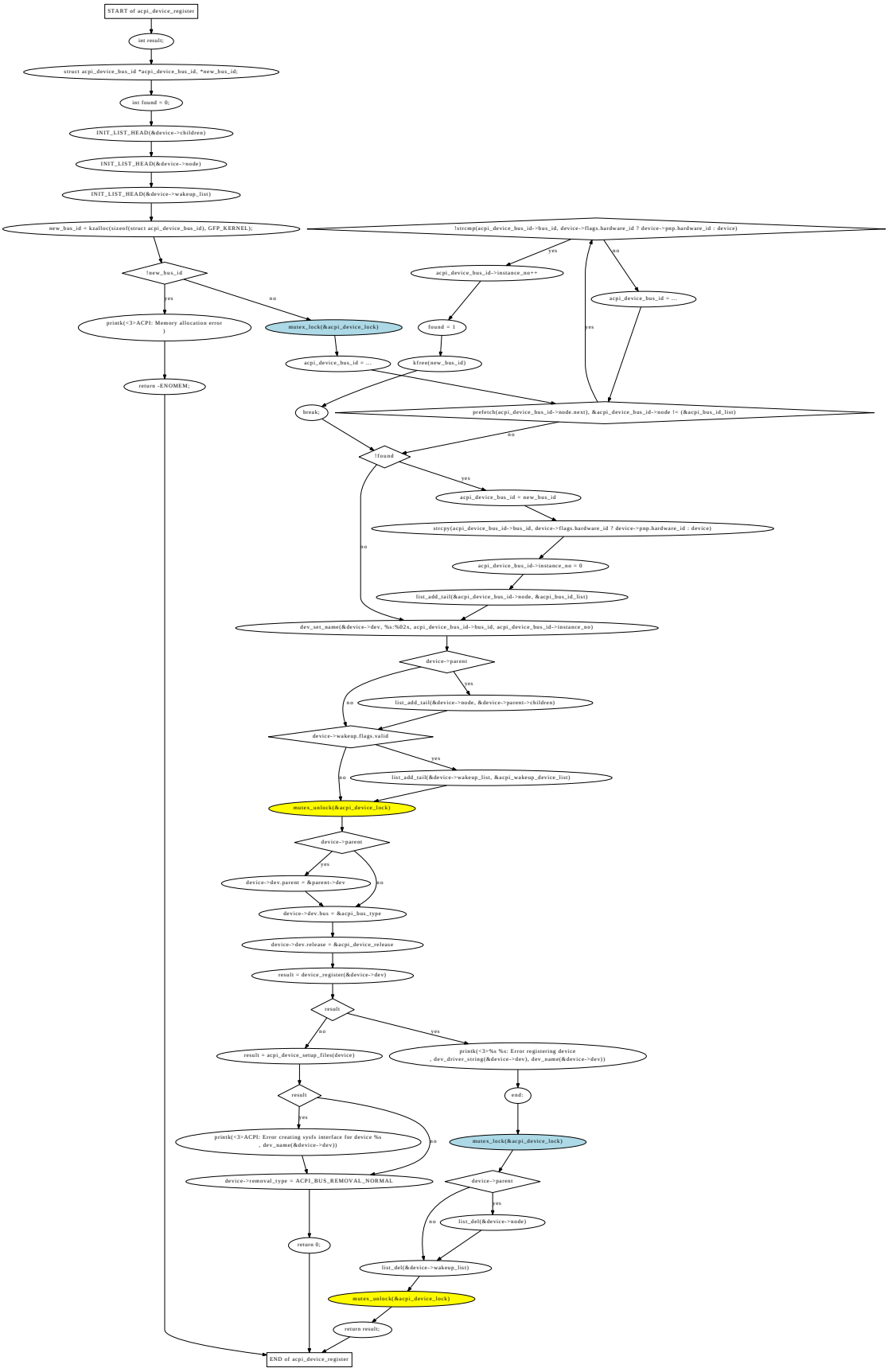


Figure 5.8 CCFG of function `acpi_device_register`

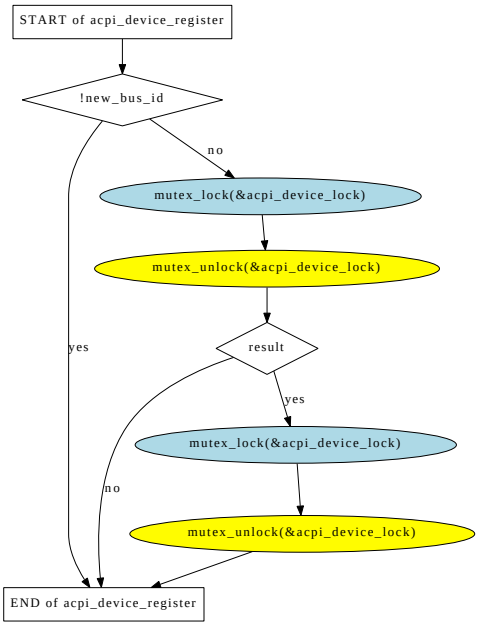


Figure 5.9 ETG of function `acpi_device_register`, compared with CCFG shown in Figure 5.8, the size is greatly reduced

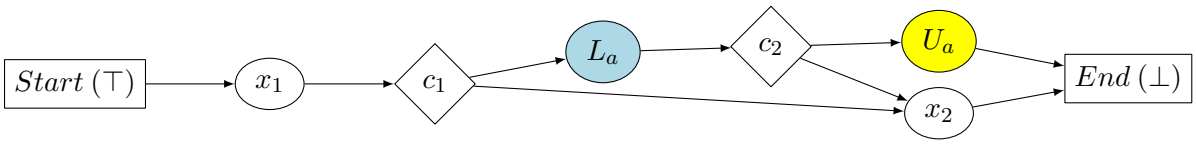


Figure 5.10 Non-structure Example

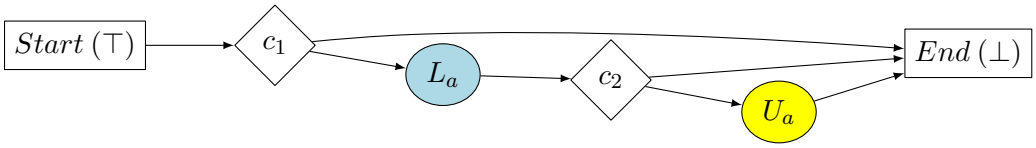


Figure 5.11 Non-structure Example Reduced

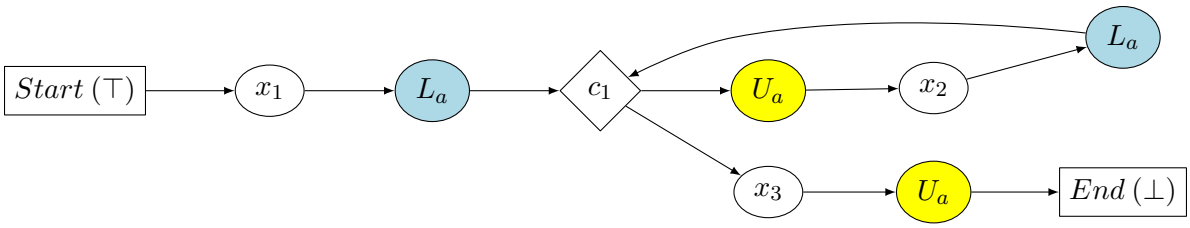


Figure 5.12 Loop example - matching property always satisfied for any numbers of loop iterations

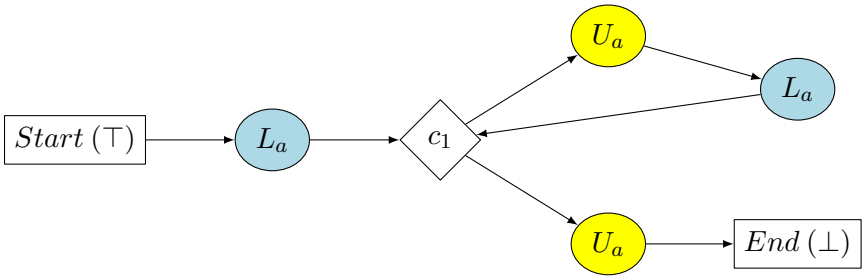


Figure 5.13 Loop example with non-important statements (x_n) removed

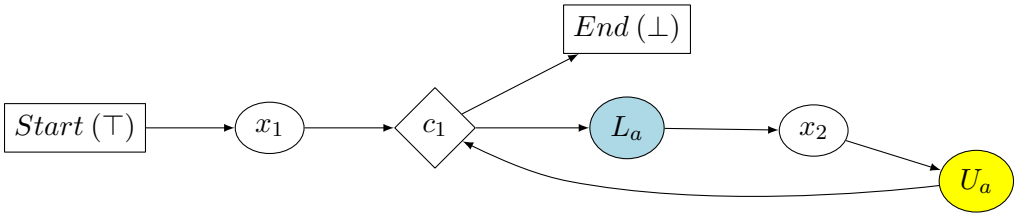
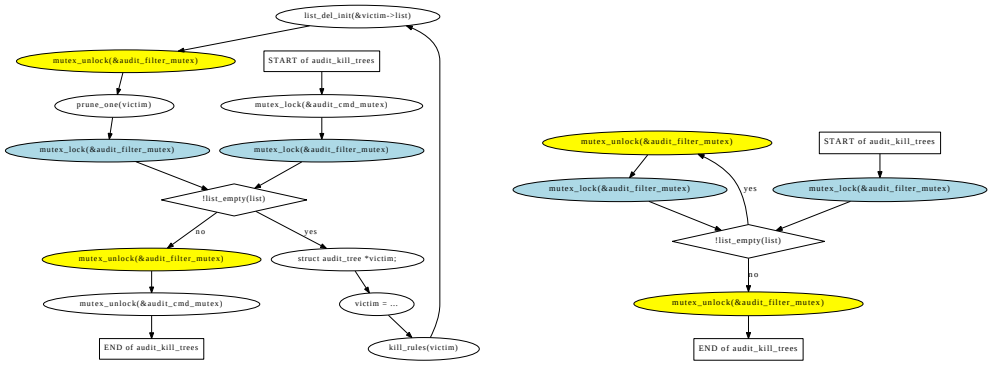


Figure 5.14 Loop example - matching property fail with 2 or more iterations



(a) CFG of loop example (b) ETG

Figure 5.15 Loop Example

5.3.1 Matching Pair Graph

Finding functions related to the event of interest is a critical part of solving the problem. For matching pair problem, event U is required to follow event L in the event trace. As shown in Figure 5.3, U events may belong to either the child function or parent function in call graph. The execution order and dependencies of functions is important to solve the problem.

Using (reverse) call graph to find the relation between function is an straight forward approach. For any L event inside function *A()*, the only functions *B()* where U belongs to, are the functions that has same common ancestor *P()* with *A()*. Function *P()* links event L and U. So for any *A()* with L insides, all its ancestor functions and the descendents of ancestor functions are the possible matching candidate. But as discussed in [19], the number of potential functions involves explodes, which many of them are not really affecting the events execution. To overcome such problem, matching pair graph (MPG) is defines for capturing only the important functions.

Figure 5.16 shows the amount of simplification lead my using MPG instead of call graph.

5.4 Proving Matching Event Properties

5.4.1 Event Signature

The first challenge of solving the matching pair problem is to identify whether two giving L and U events operate on the same object. Traditional approaches analyze the dataflow starting from L event and track the information until the object terminates or exits the analyzable scope. It produce good accuracy while suffer from two aspects, speed and scope.

These data flow analysis methods try to capture more information to make the tracking more precious, where the complexity of the method make they suffer in two areas, speed and working range. Due to the complexity of the method, inter-procedure analysis become infThere exists a faster way to solve the problem. Programmer developer uses

In general, developers probably don't use sophisticated tools to help tracking the data relation when writing code. They use some easy remember patterns. For this particular problem, developers would generally use naming patterns to related objects. By looking at

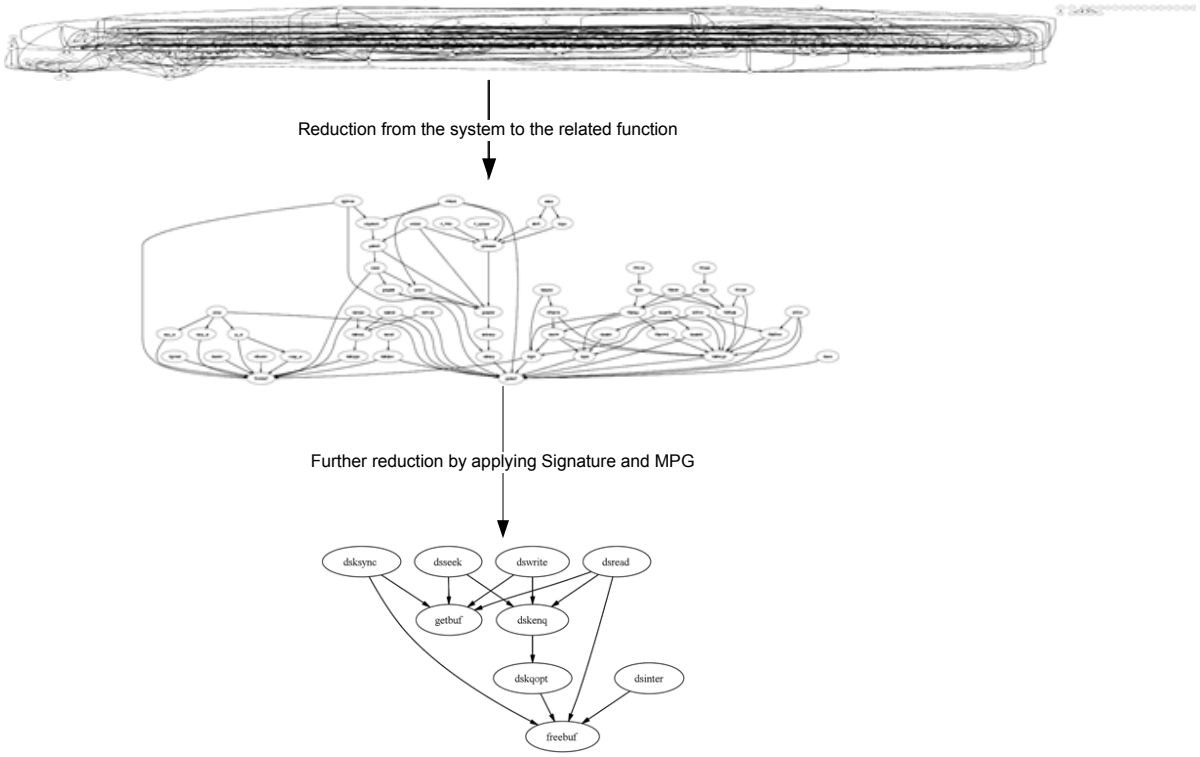


Figure 5.16 MPG greatly reduce the amount of functions involved in detailed analysis

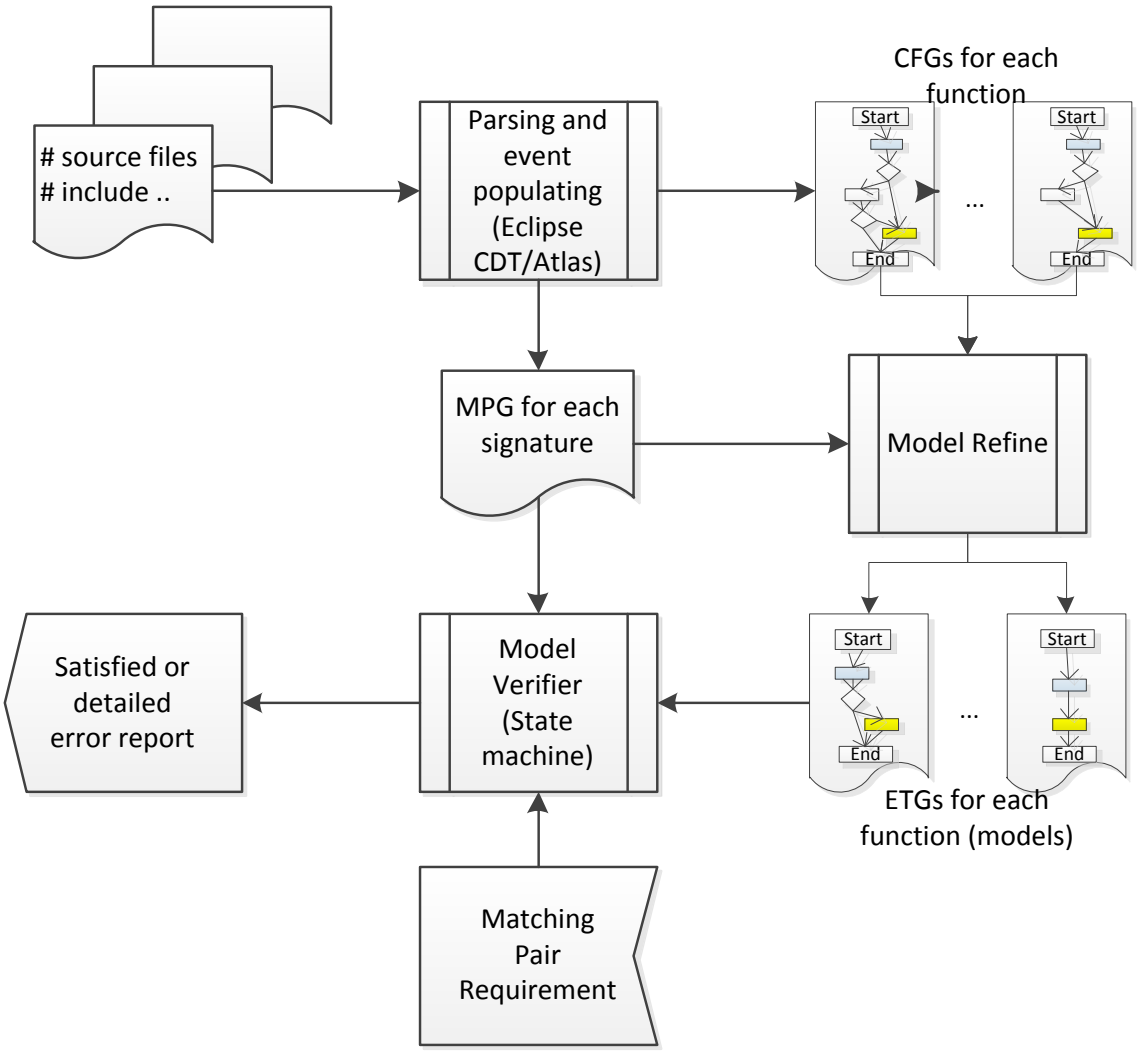


Figure 5.17 Process Flow of Event-based verification

the code itself, developer would immediate know event `mutex_lock(audit_cmd_mutex)` and `mutex_unlock(audit_filter_mutex)` don't operate the same object w/o using any special tools.

We provides two ways to identify such naming patterns.

5.4.1.1 Global Signature

As shown in Figure 5.4, `mutex_lock()` and `mutex_unlock()` accepts an argument. Further more, the argument is not some local variable, but global variable. This information provides a strong clue of which two or more events would possible operate on the same object. We think `mutex_lock()` at line 2 and line 3 are not related because `audit_cmd_mutex` and `audit_filter_mutex` two different global variables. In general, no developer would write code that mix two or more global variables together for the same object. Using global variable as the naming patten is called *global signature*.

Definition 9 *If a global variable g is used as an argument to L or U events, g itself is called a* Global Signature.

Signature can be used to group related L and U events. We say two events L and U are potentially *related* if and only if L and U has the same signature. Let's denote L_s and U_s as the L and U events with signature s. In the entire, multiple instance of L_s and U_s event may exist. Next part of paper will discuss how to combines this information with event trace to analyze the matching pair problem.

Signature approach provides a fast way of finding the relation between events. It is also capable of find potential related events across functions. Signature approach is an approximate approach, there are unavoidable limitations. Theoretically, signature can produce incorrect matching. But we think this is rear, our analysis of Linux and XINU operating systems supports out claim. (Provide more concrete number to support this argument).

```
1 static inline void backlight_update_status(struct backlight_device *bd) {
2   mutex_lock(&bd->update_lock);
3   if (bd->ops && bd->ops->update_status)
4     bd->ops->update_status(bd);
5   mutex_unlock(&bd->update_lock);
6 }
```

Figure 5.18 Type signature for mutex_lock and mutex_unlock

5.4.1.2 Container Type Signature

Definition 10 *If a variable* `a->v` *or* `a.k` *is used as the argument to L or U events,* A.v *is called a* Container Type Signature*, where A is the data type name of value a.*

Figure 5.18 shows an code example of container type signature. In this example variable `bd->update_lock` is the argument for both L and U events. We also known that the type name of `bd` is `struct backlight_device`. By definition 10, the container type signature in this example is `backlight_device.update_lock`.

5.4.2 Successor and Predecessor Pattern

From the definition of matching pair problem, successor events of an the L event are important. Event successor set is defined as the collection of all successor.

Definition 11 Successor Set*, SS_e, of event e is the set of all successor of e in all event traces.*

As we know, there are only 4 types of events defined in an event trace. Only 3 of them, L, U and $\perp$ are possible successor of any L event. Though, the event set may have large number of elements. the types of successor of an L is not larger than 3. Successor pattern is defined to represent the existence of different event types in successor set. Let's use Lk to represent whether there exists a L in the SS_e, $Lk = L$ means yes, $Lk = -$ means no. Similarly, Uk and $\perp k$ for events U and $\perp$ can be defined.

Definition 12 Successor pattern*, π_e, of an event e is a tuple $(Lk, Uk, \perp k)$, where $Lk \in \{L, -\}, Uk \in \{U, -\}, \perp k \in \{\perp, -\}$.*

$\pi_e = (-, U, -)$ means event e has one or more U successors, but no L and $\perp$ successor. $\pi_e = (L, U, \perp)$ means e has all 3 possible successors. There are 8 total successor patterns available. But since any none $\perp$ event has at least one successor event, $\pi_e \neq (-, -, -)$, if $e \neq \perp$. The number of possible π in this cases is 7.

5.4.3 Matching Difficulty Classification

Successor pattern is composed of 3 elements of binary values, there are 7 possible patterns for any none ending event. Successor pattern $(-, -, -)$ is not feasible since every none ending event has at least one successor event. These 7 patterns can be separated into 3 difficulty levels based on the amount of additional information required to answer the problem – whether U is followed by L under all feasible execution paths. Right now, 2 types of information are considered: whether the value of condition statements are important to get a conclusion; and whether invisible control flow could be involved in the problem.

Based on the above information, we classify the successor pattern into 3 difficulty levels:

Easy: Result known, condition doesn't matter

Medium: Require analyze control conditions to know result

Hard: invisible control flow may also be required to get result

The classification is based on the successor pattern by the following rules:

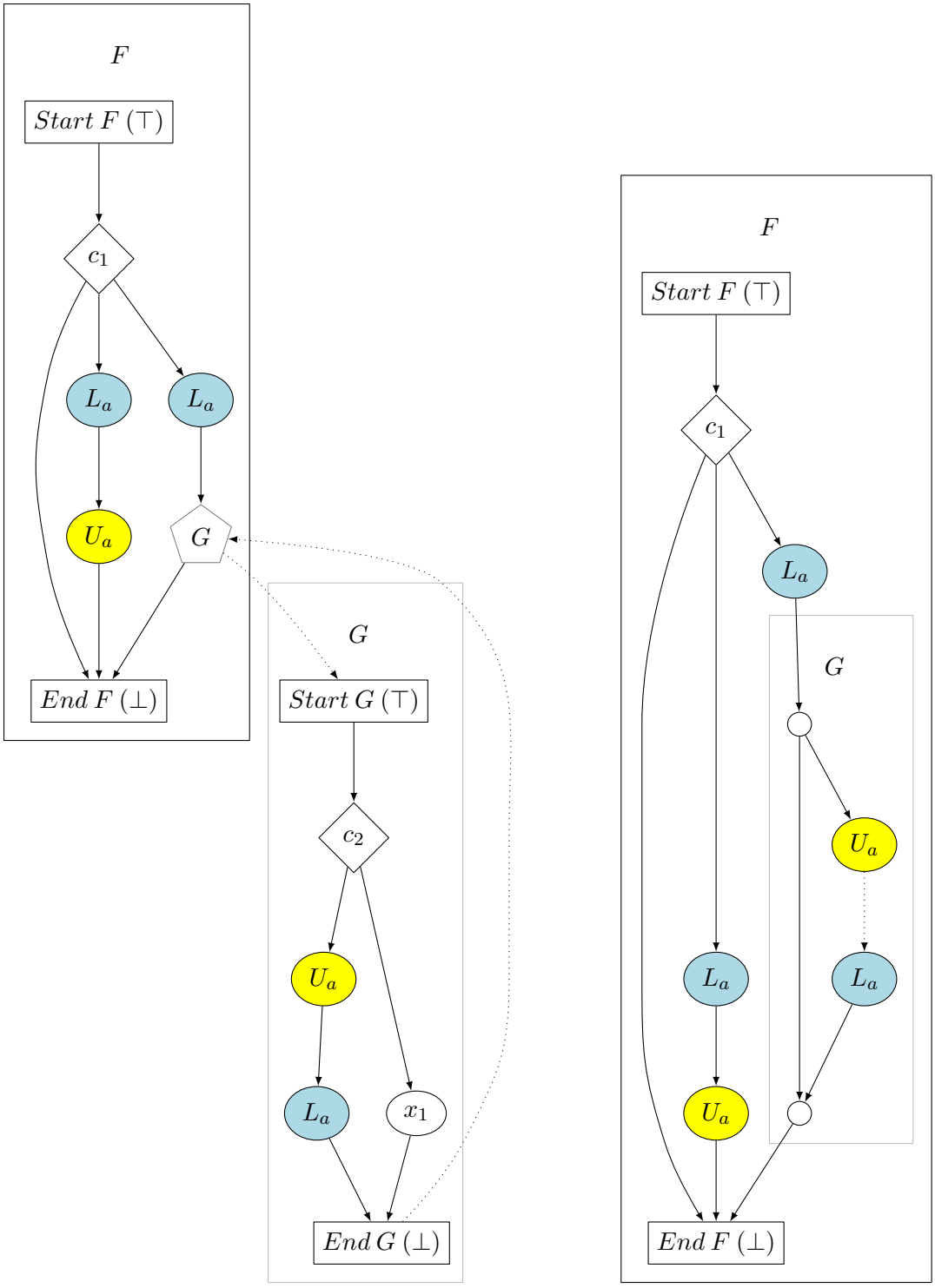
if	Only one types of event in π and its not $\perp$	easy
else if	$\perp \in \pi$ & U not in π	hard
else	-	medium

Table 5.1 shows the classification of successor patterns and their difficulty levels.

5.5 Linux Mutex Matching Evaluation

To evaluate the effectiveness and performance of signature pattern method, we applied the method on the problem of Linux mutex lock synchronization. In Linux, mutex are locked with function `mutex_lock()`[1], and unlocked by `mutex_unlock()`. These functions will corresponding

[1] In some cases, `mutex_trylock`, `mutex_lock_interruptible`, are 2 other functions are also used for locking mutex.



(a) Function F Call G, the MP property should be analyzed by combining both of them

(b) Merge G into F, none important nodes are eliminated

Figure 5.19 Inter-procedure property matching

Table 5.1 Classification based on successor pattern

#	Type	Succ. Patt.	Classification	Diffculity
1	I	(L, U, ⊥)	Possibly validated with visible flow	Medium
2		(L, U, –)		
3		(–, U, ⊥)		
4	II	(L, –, ⊥)	Invisible flow related	Hard
5		(–, –, ⊥)		
6	III	(L, –, –)	Error	Easy
7	IV	(–, U, –)	Validated	Easy

to event L and U respectfully. Global and type signatures are used to related L and U on the same object. 3 versions of Linux kernel have been tested.

Linux source code is parsed with eclipse CDT. CCFG for each function and MPG for each signature are generated with Atlas plug-in from EnSoft corp,. Later, CCFG and MPG are used together to get the ETG and eventually successor pattern for each instance of L. Predecessor pattern for all instances of U are also generated as it provides evidences for analyzing invisible control flow.

5.5.1 Linux Mutex Matching Evolution

3 versions of Linux kernel has been tested and evaluated. The evolution results are shown in Table 5.2. The number of locking events has increased significantly from 487 total in version 2.6.26 to 1251 in version 2.6.31. Of all these locking events, about 90% of the cases are able to be fully validated (type IV) by the successor pattern. As we expected, majority of the validated cases only involves intra-procedure matching, the remaining 5% are inter-procedure matching cases. We don't find any absolute error where locking is followed by another locking under all possible event tracks (type III), which is good sign. About 7% percent of the locking require further path feasibility verification (type I) to get a complete verification. Another small portion of the cases verification (type II) require invisible control flow analysis. To fully understand and verify the last two cases, manual inspection is required, but can be assisted with information provided from MPG and ETG.

Table 5.2 Validation Results for 3 versions of Linux Kernel

version	# of locks (NL)	I	II	III	IV		signature fail	Ratio of IV/NL
					intra-proc.	inter-proc.		
2.6.26 Global	257	17	4	0	231	5	0	0.92
2.6.26 Type	230	31	4	0	182	10	3	0.83
2.6.28 Global	452	37	6	0	401	8	0	0.90
2.6.28 Type	460	45	5	0	398	9	3	0.88
2.6.31 Global	621	34	10	0	562	15	0	0.93
2.6.31 Type	630	50	4	0	559	13	4	0.91

5.5.2 ETG Reduction

Compared with traditional control flow graph, instead of keep all statement and control branches, ETG only captures the important events as well the critical execution branches/control branches that may affect the order/existance of important event under event traces. It makes the graph representation easier to understand and more compact. The number of event nodes, edges as well as control nodes reduces compared with CFG. Reduction results of Linux kernels are shown in Figure 5.4.

ETG contains 70% less statement nodes and edges. More than 55% of the control statements are eliminated compared with original CFG. These result is very consistent for all 3 versions of Linux.

5.5.3 Linux Case Analysis

By going through the mutex synchronization matching of Linux kernel. We find several interesting type of matchings. Some examples are presented here.

5.5.3.1 Many to Many Event Mapping

This example in Figure 5.20 is the ETG of function `snd_timer_open()`, which shows:

- Multiple locking and unlocking events may appear in the same function. Same L can match with different U under different conditions, vice versa.
- The first event by execution function `snd_timer_open()` is either and L event or $\perp$ event.

Table 5.3 Compared with CFG, the number of nodes and edges in ETG reduced about 75%. Control statements reduced about 60% in 3 versions of Linux

version	CFG			ETG			Ratio		
	nodes (cn)	edges (ce)	ctrl. stmt. (cc)	nodes (en)	edges (ee)	ctrl. stmt. (ec)	en / cn	ee / ce	ec / cc
2.6.26 Global	7158	8273	1359	1898	2130	547	0.27	0.26	0.40
2.6.26 Type	9451	11126	1968	2371	2890	889	0.25	0.26	0.45
2.6.28 Global	13264	15383	2654	3594	4165	1147	0.27	0.27	0.43
2.6.28 Type	15254	18092	3203	3717	4540	1339	0.24	0.25	0.42
2.6.31 Global	18776	22122	3963	4857	5697	1599	0.26	0.26	0.40
2.6.31 Type	23004	27433	4880	5352	6669	2023	0.23	0.24	0.41

Table 5.4 6 examples of graph size comparison between CFG and ETG from Linux 2.6.31

Function	Signature	CFG			ETG			Ratio		
		nodes (cn)	edges (ce)	ctrl. stmt. (cc)	nodes (en)	edges (ee)	ctrl. stmt. (ec)	cn - en	ce - ee	cc - ec
snapshot_ioctl	pm_mutex	160	210	30	7	10	4	153	200	26
zisofs_readpage	zisofs_zlib_lock	173	211	39	21	37	17	152	174	22
ieee80211_register_hw	rtnl_mutex	129	157	29	16	25	10	113	132	19
migration_call	callback_mutex	118	158	32	5	5	1	113	153	31
tc_ctl_tfilter	rtnl_mutex	138	175	36	26	47	22	112	128	14
cgroup_get_sb	cgroup_mutex	109	143	35	13	19	7	96	124	28

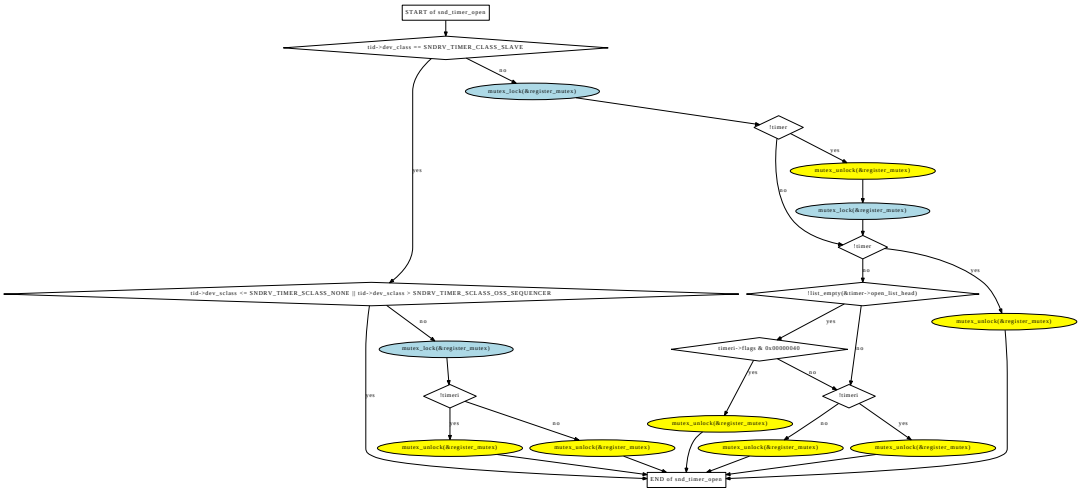


Figure 5.20 ETG of function `snd_timer_open()`

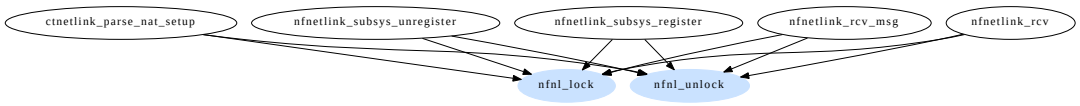


Figure 5.21 MPG of signature nfnl_mutex

5.5.3.2 Invisible Control Flow

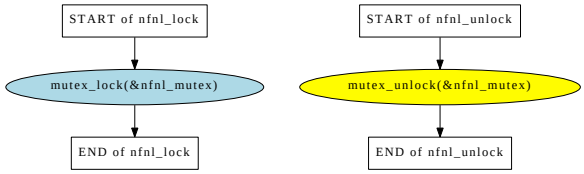
MPG of signature `infnl_mutex` is shown in Figure 5.21.

In the figures of event trace graph (ETG), square box represents the start and end of a function. Diamond represent the control conditon while eclipse represents other statement. Event are highlighted differently based on its type. L event is represented with light blue. U event is represented with yellow. To track inter-procedure matching/problen, functions may call L or U events by itself or through a series of call chains are also highlighted with shape hexagon and color light blue. As shown in Figure 5.22, the ETG of both functions `nfnl_lock()` and `nfnl_unlock()` are very simple. They call `mutex_lock()` or `mutex_unlock()` respectively, but not both.

Let's look at function `nfnetlink_rcv_msg()` of which the ETG is shown in Figure 5.23(a). This function contains two types of event traces:

1. No function call of either `nfnl_lock()` or `nfnl_unlock()`;

2. Exits function call of both `nfnl_lock()` and `nfnl_unlock()` in the order that any `nfnl_unlock()`



(a) ETG of nfnl_lock() (b) ETG of nfnl_unlock()

Figure 5.22 ETG of function nfnl_lock() and nfnl_unlock()

followed by one and only one `nfnl_lock()` call.

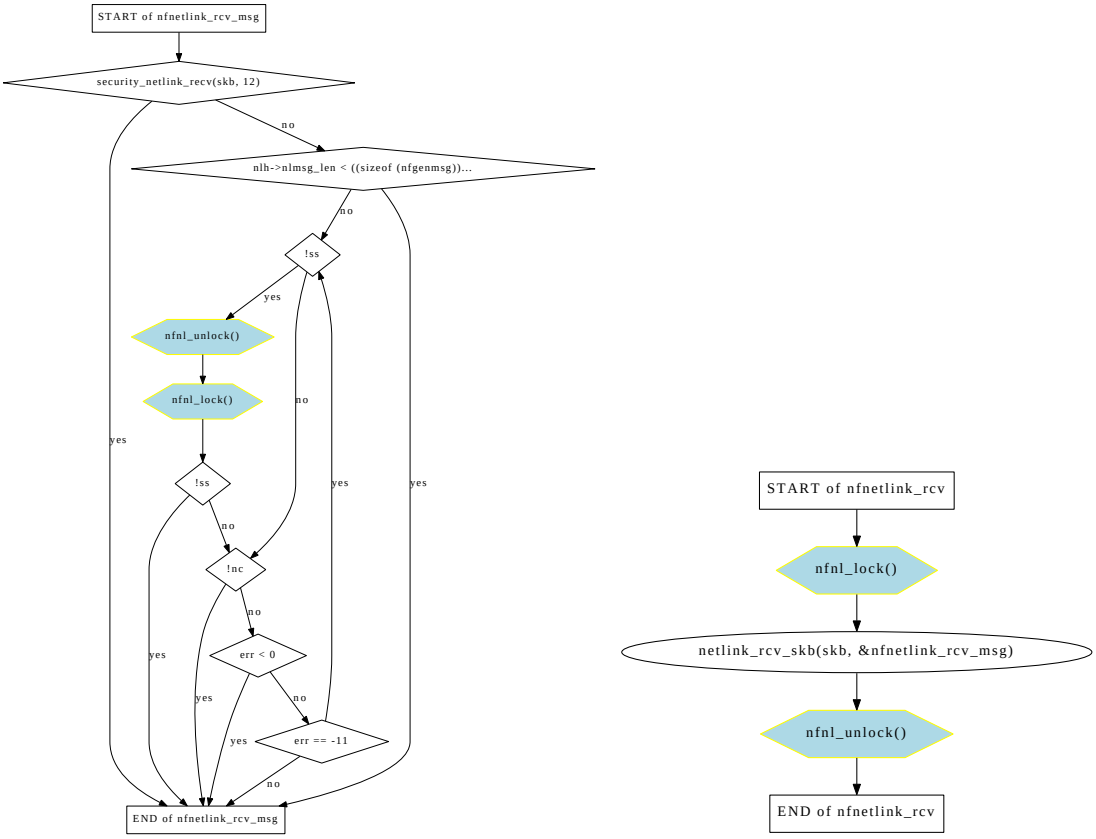
The event traces represented with regular expression would be $\top(UL)*\bot$, where $*$ represents 0 or more times.

By looking at `nfnetlink_rcv()` itself, it is not clear whether any event L will eventually match with a U event. Since there does not exist a matching U for the last L event in the event trace inside this function. Inter-procedure analysis must be involved to answer this question. But there doesn't exist any visitable function calls to `nfnetlink_rcv()` in the MPG(`nfnl_mutex`). The only possibility is there exists a invisible flow, possibly by function pointer.

By checking another function `nfnetlink_rcv()` in MPG, the desired function pointer can be found between two function calls `nfnl_lock()` and `nfnl_unlock()`. The combined event traces represented with regular expression will becomes $\top L(UL) * U\bot$. Any L event in the trace is followed by a U event. No L is left unmatched.

5.5.3.3 Infeasible Control Path

The MPG of signature `register_mutex` is shown in Figure 5.24. As we can see, there is a long chain from the top caller function `snd_seq_open()` to `snd_timer_open()` and `snd_timer_close()`. All of the 3 functions contains L event inside. By looking at the ETG of `snd_seq_open()`, we can find the event trace in which L in `snd_seq_open()` is directly followed by another L event in `snd_seq_open()` or `snd_seq_close()` through a series of functions as shown in Figure 5.24. Two consequent L events of the same object may lead to deadlock which is a serious problem. The question is whether the execution path associated with this problematic event trace is feasible?



(a) ETG of nfnetlink_rcv_msg()

(b) ETG of nfnetlink_rcv()

Figure 5.23 nfnetlink_rcv() and nfnetlink_rcv_msg()

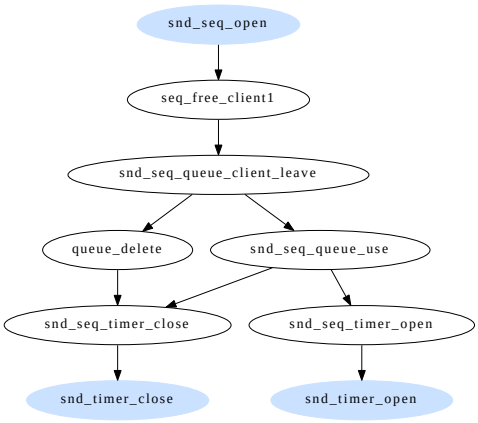
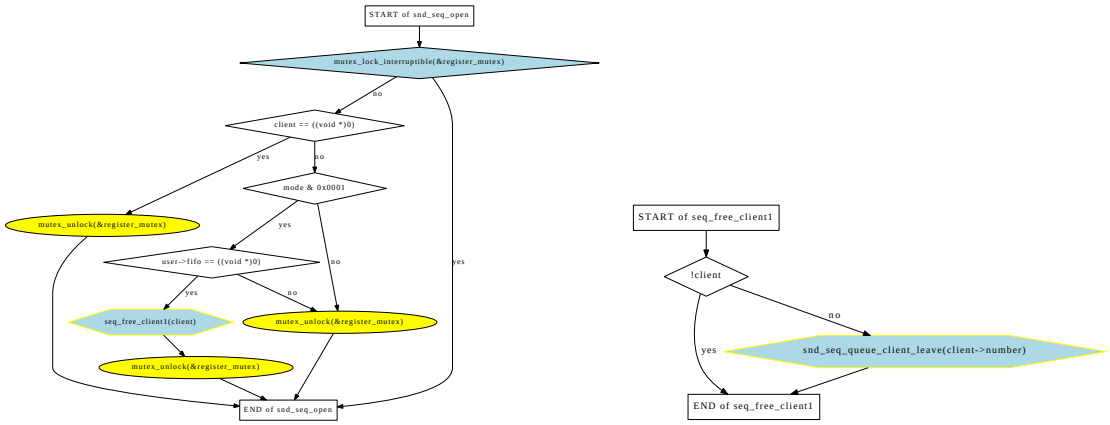


Figure 5.24 MPG of signature register_mutex

This question can not be answered automatically through our successor pattern. But using the critical control condition provided in ETG of `snd_seq_open()` shown in Figure 5.25(a) and its child `seq_free_client1()` shown in Figure 5.25(b). We found contradiction conditions. In `snd_seq_open()`, for function `seq_free_client1()` to be invoked, condition `client == ((void *)0)` must evaluate to `false`. But inside function `seq_free_client1()`, only when `!client` evaluate to `false` will `snd_seq_queue_client_leave()` be invoked. Combine the information above, we may get the conclusion that `snd_seq_queue_client_leave()` will never be invoked through `snd_seq_open()`. As we can see in the MPG of signature `register_mutex`, without `snd_seq_queue_client_leave()`, there will be no possible path from `snd_seq_open()` to either `snd_timer_open()` or `snd_timer_close()`, which means no possible double lock of the same object.

5.6 Conclusion and Future Work

Matching pair problems like mutex synchronization has been difficulty problem for some time []. Research has applied approaches like static analysis and model based verification to the problem, but these approaches all facing the some problems especially the ability to handle matching across multiple functions or with invisible control flow. In this paper, we has discussed a different approach which scales to large system with complex matching possible. We first discussed the model and general model for validate matching pair problem. Later, we discussed



(a) ETG of snd_seq_open() (b) ETG of function seq_free_client1()

Figure 5.25 snd_seq_open() and seq_free_client1()

the notion of event and event traces to represent the execution order of the important actions during different program executions. To trace more precise information, the same action over different object instance are consider different. Signatures were defined as an approximation to differ instances. Successor pattern was latter introduced to properly handle multiple event traces as the same time during the analysis. With the above techniques, matching within a function could be solved already. But there still remains portion of the problems requires further analysis. This remain part generally involves inter-process behavior or even invisible control, To handles these properly, we discussed Matching Pair Graph as well as extending the Event Trace Graph for multiple functions.

We have applied this techniques on 3 versions of Linux kernels and try to analyze the mutex synchronization problem thoroughly. Over more than 2600 cases, our approach solves more than 90% of them, in the remaining of the cases, which requires further analysis of the execution condition, our ETG provides informative results which generally remove 60% of the conditions which are irrelevant to the answer.

For future works, there are multiple paths to go. For unsolved the problems, validation becomes the satisfiability problem of critical control conditions remain the ETG. Right now, we solve it manually, but automatic process could be added to make the analysis more complete. ETG discard non-critical information from the general graph representations like control flow

graph. It can be applied to many other problems besides the matching pair problem. Using it as the platform to solve other event trace based problem will make it more interesting.

CHAPTER 6. SUMMARY AND CONTRIBUTION

The main contributions of this dissertation are:

- **Matching Pair Graph:** The difficulty of testing and other static program analysis techniques are for complex system which may involve large number of functions. Matching Pair Graph is designed to overcome such limitation. It greatly reduces the number of functions to be analyzed for proving the matching pair property.

- **Event Trace Graph:** Path explosion is another difficulty for the program analysis. The number of execution traces increase exponentially as the number of control statements increases, the complexity to solve such problem increases accordingly. Event Trace Graph is designed to address this difficulty. Only a subset of control statement which are critical to the event trace are kept, while majority of non-related branches are eliminated. we were able to prove the correctness of more than 90% of the synchronization instances in the Linux kernel. For each remaining case, we produced relevant ETGs for the further investigation by human experts.

- **Signatures:** Tracking data flow information is another challenging problem. Unlike compilers developers use patterns to track the data. We introduced the notion of signature which is similar to the patterns used developer.

- **Empirical Study of Matching Pair Property of Linux:** Mutex matching pair property has been studied and discussed in this research. 8 versions of Linux kernels spanning over 3 years has been studied and examined. 3 versions has been verified in more detail as the ETG for each function has been generated and verified. This is the first research of its kind.

APPENDIX A. LIST OF SIGNATURES AND THEIR MATCHING PAIR PROPERTIES

Due to the limitation of space, the detailed signature information from only one version of Linux (2.6.31) is listed. All other versions analyzed have the similar report. Description of Table A.1 are:

ID: ID of each signature, count separately for global and type signatures

Signature Name: Name of the signature

Type: Type of the signature. **G** for global, **T** for type

N_L: Number of functions contains L events (`mutex_lock` etc.)

N_U: Number of functions containing U events (`mutex_unlock`)

RCG-C: Size of the reverse call graph

MPG-BAL: Number of balanced nodes

UBC: Number of UBC nodes

MPG: Number of MPG nodes

Table A.1: Detailed information about signature and MPG of Linux version 2.6.31

ID	Sigature Name	Type	N_L	N_U	RCG-C	MPG-BAL	UBC	MPG
1	_event_lock	G	2	2	50	2	2	2
2	acpi_device_lock	G	3	3	24	3	3	3
3	acpi_link_lock	G	4	4	8	4	4	4
4	afinfo_mutex	G	2	2	8	2	2	2
5	all_stat_sessions_mutex	G	2	2	2	2	2	2
6	allocated_ptys_lock	G	2	2	8	2	2	2
7	attribute_container_mutex	G	6	6	87	6	6	6
8	audit_cmd_mutex	G	3	3	3	3	3	3

9	audit_filter_mutex	G	14	14	88	14	14	14
10	bio_slab_lock	G	2	2	66	2	2	2
11	blk_tree_mutex	G	1	1	1	1	1	1
12	block_class_lock	G	3	3	17	3	3	3
13	br_ioctl_mutex	G	1	1	1	1	1	1
14	brd_devices_mutex	G	1	1	1	1	1	1
15	bsg_mutex	G	6	6	62	6	6	6
16	btrace_mutex	G	1	1	2	1	1	1
17	callback_mutex	G	11	11	21	11	13	13
18	cdrom_mutex	G	3	3	5	3	3	3
19	cfg80211_mutex	G	10	10	60	10	10	10
20	cgroup_mutex	G	13	13	34	23	25	25
21	chrdevs_lock	G	3	3	63	3	3	3
22	cm_sbs_mutex	G	4	4	8	4	4	4
23	con_buf_mtx	G	1	1	1	1	1	1
24	core_lock	G	9	9	36	9	9	9
25	cpu_add_remove_lock	G	1	1	228	8	10	12
26	cpufreq_governor_mutex	G	3	3	10	3	3	3
27	cpuidle_lock	G	2	2	12	3	5	5
28	crypto_default_rng_lock	G	2	2	6	2	2	2
29	dbs_mutex	G	5	5	5	5	5	5
30	dcookie_mutex	G	3	3	3	3	3	3
31	device_add_lock	G	1	1	7	1	1	1
32	dlci_ioctl_mutex	G	1	1	1	1	1	1
33	dnotify_mark_mutex	G	2	2	132	2	2	2
34	dpm_list_mtx	G	9	9	900	9	11	11
35	dst_gc_mutex	G	2	2	2	2	2	2
36	epmutex	G	2	2	6	2	2	2
37	evdev_table_mutex	G	2	2	5	2	2	2
38	event_mutex	G	7	6	14	5	8	8
39	ext_devt_mutex	G	2	2	83	2	2	2
40	fsnotify_grp_mutex	G	2	3	8	2	4	4
41	fw_lock	G	4	4	26	4	4	4
42	genl_mutex	G	1	1	25	8	10	10

43	global_host_template_mutex	G	2	2	19	2	2	2
44	hash_mutex	G	2	2	4	2	2	2
45	hash_resize_mutex	G	2	2	2	2	2	2
46	hid_open_mut	G	2	2	6	2	2	2
47	host_cmd_pool_mutex	G	2	2	20	2	2	2
48	idr_lock	G	2	2	5	2	2	2
49	info_mutex	G	5	5	103	5	5	5
50	input_mutex	G	6	6	55	4	8	8
51	iprune_mutex	G	2	2	112	2	2	2
52	kexec_mutex	G	2	2	1103	2	2	2
53	key_construction_mutex	G	2	2	37	2	2	2
54	key_mutex	G	1	1	67	2	4	4
55	key_session_mutex	G	1	1	1	1	1	1
56	key_user_keyring_mutex	G	1	1	14	1	1	1
57	kprobe_insn_mutex	G	2	2	20	2	2	2
58	kprobe_mutex	G	9	9	21	9	9	9
59	list_mutex	G	4	4	16	4	4	4
60	lock	G	1	1	44	1	1	1
61	loop_devices_mutex	G	1	1	1	1	1	1
62	markers_mutex	G	4	4	7	4	4	6
63	microcode_mutex	G	3	3	4	3	3	3
64	minors_lock	G	3	3	6	3	3	3
65	misc_mtx	G	4	4	46	3	5	5
66	mm_all_locks_mutex	G	1	1	2	1	2	2
67	module_mutex	G	5	5	113	4	6	6
68	mon_lock	G	4	4	6	4	4	4
69	mousedev_table_mutex	G	2	2	9	2	2	2
70	mtrr_mutex	G	2	2	29	2	2	2
71	net_mutex	G	11	11	121	11	11	11
72	nf_ct_ext_type_mutex	G	2	2	16	2	2	2
73	nf_ct_helper_mutex	G	2	2	8	2	2	2
74	nf_ct_proto_mutex	G	4	4	8	4	4	4
75	nf_hook_mutex	G	2	2	23	2	2	2
76	nf_log_mutex	G	5	5	14	5	5	5

77	nf_sockopt_mutex	G	3	3	9	3	3	3
78	nfnl_mutex	G	1	1	14	5	7	7
79	nfs_callback_mutex	G	2	2	115	2	2	2
80	nlm_file_mutex	G	3	3	58	3	3	3
81	nlm_host_mutex	G	3	3	31	3	3	3
82	nlmsvc_mutex	G	2	2	13	2	2	2
83	notify_lock	G	2	2	7	2	2	2
84	nvm_mutex	G	1	2	7	5	6	6
85	ops_mutex	G	6	6	89	6	6	6
86	osc_lock	G	2	2	7	2	2	2
87	pci_hp_mutex	G	2	2	7	2	2	2
88	pci_remove_rescan_mutex	G	3	3	3	3	3	3
89	pcpu_alloc_mutex	G	2	2	544	2	2	2
90	percpu_counters_lock	G	3	3	50	3	3	3
91	performance_mutex	G	4	4	8	4	4	4
92	phy_fixup_lock	G	2	2	18	2	2	2
93	pid_caches_mutex	G	1	1	7	1	1	1
94	pm_mutex	G	9	9	14	9	9	9
95	pmc_reserve_mutex	G	1	2	4	1	2	2
96	pnp_res_mutex	G	2	2	8	2	2	2
97	polldev_mutex	G	2	2	4	2	2	2
98	pools_lock	G	3	3	19	3	3	3
99	port_mutex	G	2	2	14	2	2	2
100	preset_mutex	G	3	3	29	3	3	3
101	probing_active	G	1	2	8	2	5	5
102	profile_flip_mutex	G	2	2	2	2	2	2
103	psmouse_mutex	G	7	7	7	7	7	7
104	queue_handler_mutex	G	3	3	3	3	3	3
105	rate_ctrl_mutex	G	3	3	13	3	3	3
106	rcu_barrier_mutex	G	1	1	210	1	1	1
107	register_mutex	G	24	24	63	24	24	32
108	relay_channels_mutex	G	6	6	17	6	6	6
109	rfkill_global_mutex	G	11	11	27	11	11	11
110	rng_mutex	G	5	5	13	5	5	5

111	rsrc_mutex	G	9	9	13	9	9	9
112	rtnl_mutex	G	2	1	188	93	97	98
113	sched_domains_mutex	G	2	2	10	2	2	2
114	sched_register_mutex	G	4	4	17	4	4	4
115	sd_ref_mutex	G	4	4	10	4	4	5
116	serial_mutex	G	2	2	10	2	2	2
117	serio_mutex	G	5	5	11	5	5	5
118	setup_lock	G	2	2	44	2	2	2
119	shares_mutex	G	1	1	2	1	1	1
120	shmem_swaplist_mutex	G	3	4	5	3	4	4
121	show_mutex	G	1	1	1	1	1	1
122	smp_alt	G	3	3	7	3	3	3
123	snd_card_mutex	G	9	9	99	9	9	9
124	sound_mutex	G	5	5	19	5	5	5
125	sr_ref_mutex	G	3	3	5	3	3	3
126	strings	G	2	2	13	2	2	2
127	svc_pool_map_mutex	G	3	3	132	3	3	3
128	swapon_mutex	G	1	1	2	0	2	2
129	sysdev_drivers_lock	G	5	5	58	5	5	5
130	sysfs_bin_lock	G	3	3	530	3	3	3
131	sysfs_mutex	G	9	10	544	15	16	16
132	sysfs_rename_mutex	G	3	3	4	3	3	3
133	sysfs_workq_mutex	G	2	2	2	2	2	2
134	text_mutex	G	7	7	33	7	7	9
135	therm_cpu_lock	G	2	2	2	2	2	2
136	thermal_list_lock	G	5	5	21	5	5	5
137	trace_types_lock	G	11	11	13	10	12	12
138	tracepoints_mutex	G	6	6	114	6	6	8
139	tty_mutex	G	7	7	20	6	8	8
140	tunnel4_mutex	G	2	2	4	2	2	2
141	usb_bus_list_lock	G	7	7	9	7	7	7
142	usbfs_mutex	G	2	2	2	2	2	2
143	usblp_mutex	G	3	3	3	3	3	3
144	userspace_mutex	G	2	2	2	2	2	2

145	usu_probe_mutex	G	2	2	2	2	2	2
146	vlan_ioctl_mutex	G	1	1	1	1	1	1
147	xfrm_cfg_mutex	G	0	0	0	0	0	0
148	zisofs_zlib_lock	G	1	1	1	1	1	1
1	Scsi_Host	T	9	9	47	9	9	9
2	acpi_battery	T	3	3	14	3	3	3
3	acpi_ec	T	5	5	19	5	5	6
4	acpi_power_resource	T	5	5	34	5	5	5
5	acpi_video_bus	T	3	3	7	3	3	3
6	aer_rpc	T	1	1	1	1	1	1
7	agp_front_data	T	4	4	4	4	4	4
8	ath5k_softc	T	7	7	9	7	7	7
9	atkbd	T	1	1	1	1	1	1
10	autofs_sb_info	T	6	6	19	6	6	6
11	azx	T	2	2	2	2	2	2
12	backlight_device	T	9	9	17	9	9	9
13	bin_buffer	T	1	1	1	1	1	1
14	block_device	T	12	12	91	12	12	12
15	cfg80211_registered_device	T	7	6	50	39	43	43
16	cgroup_subsys	T	2	2	9	3	5	5
17	class_private	T	4	4	889	4	4	4
18	cpu_dbs_info_s	T	2	2	2	2	2	2
19	dock_station	T	3	3	21	3	3	3
20	dquot	T	3	3	10	3	3	3
21	drm_device	T	81	82	120	81	82	83
22	drm_mode_config	T	29	29	125	29	29	46
23	elevator_queue	T	3	3	72	3	3	3
24	evdev	T	5	5	9	5	5	5
25	eventpoll	T	2	2	4	2	2	2
26	ext3_inode_info	T	4	4	42	4	4	6
27	fb_info	T	2	3	21	2	3	3
28	ff_device	T	3	3	3	3	3	3
29	fsnotify_group	T	4	4	155	4	4	4
30	hda_bus	T	3	3	604	3	3	4

31	hda_codec	T	20	20	81	20	20	20
32	hiddev	T	2	2	3	2	2	2
33	i2c_adapter	T	2	2	127	2	2	3
34	idmap	T	3	3	39	3	3	3
35	ieee80211_local	T	19	19	63	19	19	22
36	inode	T	101	105	1273	115	136	154
37	inode_security_struct	T	1	1	10	1	1	1
38	inotify_handle	T	11	11	225	11	11	11
39	input_dev	T	8	8	60	8	8	8
40	journal_t	T	3	3	25	8	11	18
41	kcopyd_job	T	1	1	6	1	1	1
42	key_user	T	1	1	7	1	1	1
43	kobj_map	T	3	3	158	3	3	3
44	loop_device	T	4	5	5	4	5	5
45	mapped_device	T	3	3	6	3	3	3
46	mddev_t	T	4	3	15	11	14	14
47	mii_bus	T	4	4	23	4	4	4
48	mm_context_t	T	2	2	20	2	2	2
49	mon_reader_bin	T	3	3	3	3	3	3
50	mousedev	T	5	5	16	5	5	7
51	msdos_sb_info	T	1	1	24	3	5	5
52	netlink_sock	T	2	2	8	2	2	2
53	nlm_file	T	4	4	55	4	4	4
54	nlm_host	T	1	1	19	1	1	1
55	ops_list	T	4	4	7	4	4	4
56	packet_sock	T	3	3	5	3	3	3
57	pci_vpd_pci22	T	2	2	2	2	2	2
58	pcmcia_socket	T	15	15	21	15	15	15
59	perf_counter	T	5	6	80	5	6	6
60	perf_counter_context	T	6	6	77	6	6	6
61	phy_device	T	11	11	34	11	11	12
62	prop_descriptor	T	1	1	2	1	1	1
63	ps2dev	T	2	2	81	2	2	2
64	quota_info	T	13	13	158	13	13	13

65	request_queue	T	3	3	69	3	3	3
66	rfkill_data	T	5	5	33	5	5	5
67	ring_buffer	T	1	1	4	1	1	1
68	rtc_device	T	9	9	23	9	9	10
69	seq_file	T	1	1	1	1	1	1
70	serio	T	6	5	17	4	7	7
71	snd_card	T	1	1	2	0	2	2
72	snd_hwdep	T	2	2	2	2	2	2
73	snd_info_entry	T	1	1	1	1	1	1
74	snd_mixer_oss	T	2	2	2	2	2	2
75	snd_pcm	T	4	4	6	4	4	4
76	snd_pcm_oss_runtime	T	2	2	19	2	2	3
77	snd_pcm_oss_stream	T	3	3	4	3	3	3
78	snd_seq_client	T	4	4	7	4	4	4
79	snd_seq_queue	T	1	1	5	1	1	1
80	snd_timer_user	T	1	1	1	1	1	1
81	spi_transport_attrs	T	1	1	3	1	1	1
82	srcu_notifier_head	T	2	2	12	2	2	2
83	srcu_struct	T	1	1	14	1	1	1
84	stat_session	T	3	3	6	2	4	4
85	super_block	T	2	2	47	25	29	29
86	superblock_security_struct	T	2	2	6	2	2	2
87	svc_xprt	T	1	1	4	1	1	1
88	task_struct	T	3	3	5	3	3	3
89	thermal_zone_device	T	3	3	13	3	3	3
90	trace_iterator	T	2	2	2	2	2	2
91	tty_audit_buf	T	5	5	72	5	5	5
92	tty_port	T	2	2	2	2	2	2
93	tty_struct	T	26	25	157	27	29	32
94	uart_state	T	14	15	38	15	15	15
95	unix_sock	T	4	4	8	4	4	4
96	us_data	T	7	7	11	6	8	8
97	usb_device	T	1	1	2	0	2	2
98	usb_hub	T	2	2	32	2	2	2

99	usblp	T	5	5	5	5	5	5
100	xt_af	T	12	14	85	11	15	15

APPENDIX B. COMPLETE LIST OF MATCHING PAIR PROPERTY PROOFING RESULT

A web site containing the complete result of 3 versions of Linux (2.26, 2.28, 2.31) has been build and hosted. This site is constructed to help research and developer quickly read, understand and prove the results provided my this research. Information are organized for different versions of Linux. For each version, a complete list of signatures are provided based on its type. On the paper designed for each signature, the Matching Pair Graph is shown along with a table containing the list of all functions in MPG. A snapshot of the CFG and the ETG for each function are also presented. By clicking the snapshot, a larger and more readable figure will shown. To further help understand the event flow, source file name and path are provided. Source code can be viewed by click on the file name.

An serial of example pages of the web site are shown blow.

Link to the entire web site: http://dl.dropbox.com/u/35447145/index.html

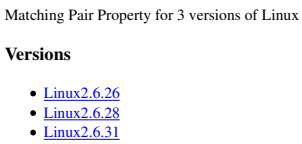


Figure B.1 Main index

All > Linux 2.6.31

globals

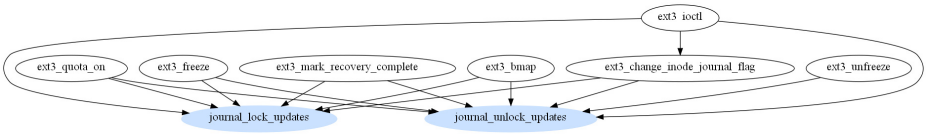
- acpi_device_lock
- acpi_link_lock
- active_counters
- afinfo_mutex
- agp_fe.agp_mutex
- allocated_ptys_lock
- all_stat_sessions_mutex
- attribute_container_mutex
- audit_cmd_mutex
- audit_filter_mutex
- bio_slab_lock
- blk_tree_mutex
- block_class_lock
- brd_devices_mutex
- br_ioctl_mutex
- bsg_mutex
- btrace_mutex
- callback_mutex
- cdrom_mutex
- cfg80211_mutex
- cgroup_mutex
- chrdevs_lock
- cm_sbs_mutex
- con_buf_mtx
- core_lock
- cpufreq_governor_mutex
- cpuidle_lock
- cpu_add_remove_lock
- cpu_hotplug.lock
- crypto_default_rng_lock
- dbs_mutex
- dcookie_mutex
- device_add_lock
- dlci_ioctl_mutex
- dnotify_mark_mutex
- dpm_list_mtx
- dst_gc_mutex
- epmutex
- evdev_table_mutex
- event_mutex
- ext_devt_mutex
- fsnotify_grp_mutex
- fw_lock
- genl_mutex
- global_host_template_mutex
- hash_mutex
- hash_resize_mutex
- hid_open_mut
- host_cmd_pool_mutex
- idr_lock
- info_mutex
- input_mutex
- iprune_mutex
- kexec_mutex
- key_construction_mutex
- key_mutex
- key_session_mutex
- key_user_keyring_mutex
- kprobe_insn_mutex
- kprobe_mutex
- list_mutex
- lock
- loop_devices_mutex
- markers_mutex
- microcode_mutex
- minors_lock
- misc_mtx
- mm_all_locks_mutex
- module_mutex
- mon_lock
- mousedev_mix.mutex
- mousedev_table_mutex
- mtrr_mutex
- net_mutex
- nfnl_mutex

Figure B.2 List of all signatures for an version of Linux, global signature and type signature are listed separately

All > Linux 2.6.31 > journal_t.j_barrier

Signature: journal_t.j_barrier

Matching Pair Graph (MPG)



MPG Function Information

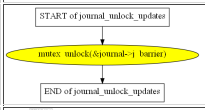
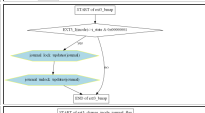
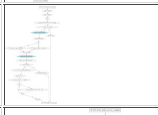
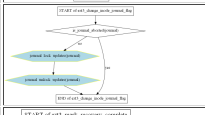
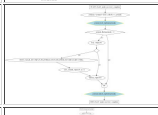
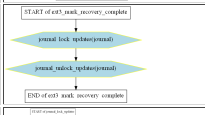
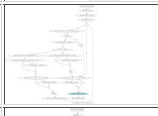
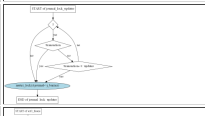
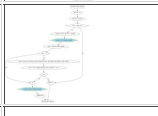
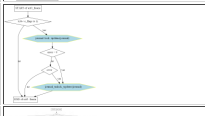
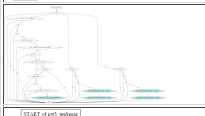
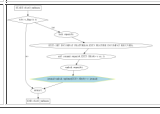
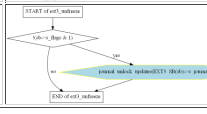
Function Name	Control Flow Graph (CFG)	Event Trace Graph (ETG)	Source Code
journal_unlock_updates			linux_2.6.31\fs\jbd\transaction.c
ext3_quota_on			linux_2.6.31\fs\ext3\super.c
ext3_bmap			linux_2.6.31\fs\ext3\inode.c
ext3_change_inode_journal_flag			linux_2.6.31\fs\ext3\inode.c
ext3_mark_recovery_complete			linux_2.6.31\fs\ext3\super.c
journal_lock_updates			linux_2.6.31\fs\jbd\transaction.c
ext3_freeze			linux_2.6.31\fs\ext3\super.c
ext3_ioctl			linux_2.6.31\fs\ext3\ioctl.c
ext3_unfreeze			linux_2.6.31\fs\ext3\super.c

Figure B.3 Main page for a signature, MPG for the signature as well as the CFG and ETG for each function in MPG are listed in a table. Link to the source code are also listed

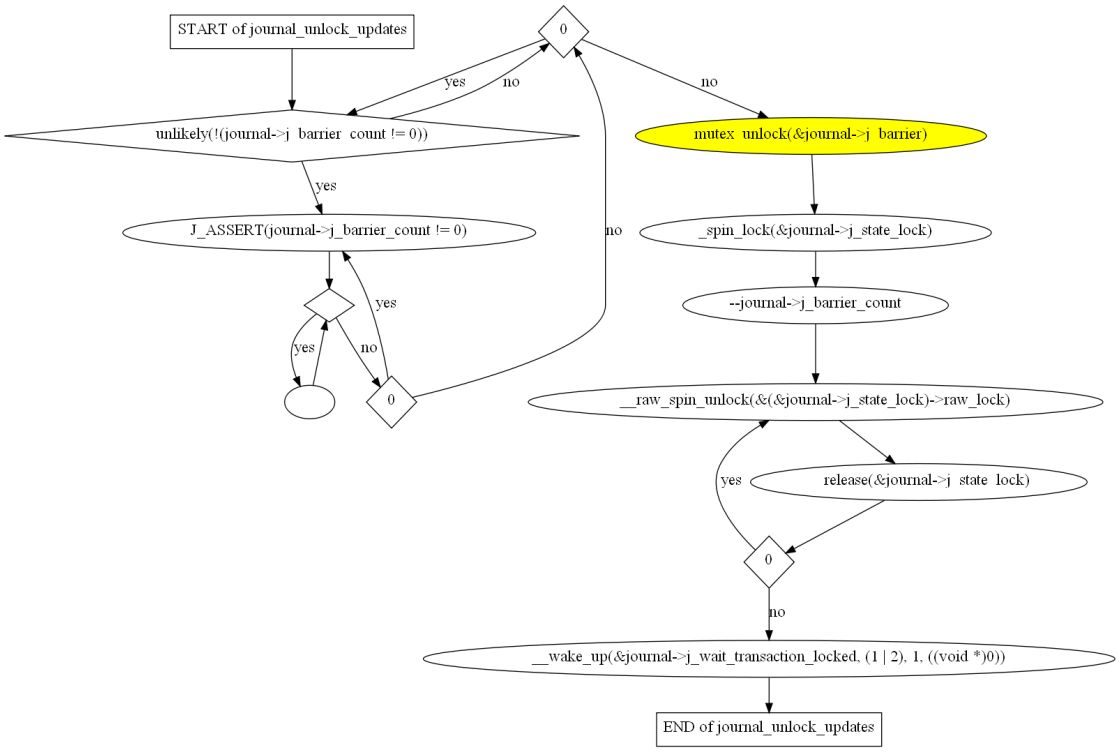


Figure B.4 Control flow graph example after click the CFG link for any function

BIBLIOGRAPHY

[1] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001). Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA. ACM.

[2] Ball, T. and Rajamani, S. K. (2002). The slam project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3.

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 196–207, New York, NY, USA. ACM.

[4] Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. (2004). Modular verification of software components in c. *Software Engineering, IEEE Transactions on*, 30(6):388 – 402.

[5] Chaki, S., Clarke, E. M., Ouaknine, J., Sharygina, N., and Sinha, N. (2005). Concurrent software verification with states, events, and deadlocks. *Formal Asp. Comput.*, 17(4):461–483.

[6] Cherem, S., Princehouse, L., and Rugina, R. (2007a). Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 480–491, New York, NY, USA. ACM.

[7] Cherem, S., Princehouse, L., and Rugina, R. (2007b). Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not.*, 42(6):480–491.

[8] Coverity. Coverity prevent. http://www.coverity.com/products/coverity-prevent.html.

[9] Das, M., Lerner, S., and Seigle, M. (2002). Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA. ACM.

[10] DeMartini, C., Iosif, R., and Sisto, R. (1999). A deadlock detection tool for concurrent java programs. *Softw. Pract. Exper.*, 29(7):577–603.

[11] Dowson, M. (1997). The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84.

[12] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515.

[13] Engler, D., Chelf, B., Chou, A., and Hallem, S. (2000a). Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA. USENIX Association.

[14] Engler, D., Chelf, B., Chou, A., and Hallem, S. (2000b). Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–1.

[15] ENSOFT. EnSoft Corp. http://www.ensoftcorp.com.

[16] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51.

[17] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

[18] Flanagan, C. and Qadeer, S. (2002). Predicate abstraction for software verification. *SIGPLAN Not.*, 37(1):191–202.

[19] Gui, K. and Kothari, S. (2010). A 2-phase method for validation of matching pair property with case studies of operating systems. In *ISSRE*, pages 151–160. IEEE Computer Society.

[20] Hausler, P., Pleszkoch, M., Linger, R., and Hevner, A. (1990). Using function abstraction to understand program behavior. *Software, IEEE*, 7(1):55 –63.

[21] Heine, D. L. and Lam, M. S. (2003). A practical flow-sensitive and context-sensitive c and c++ memory leak detector. *SIGPLAN Not.*, 38(5):168–181.

[22] Heine, D. L. and Lam, M. S. (2006). Static detection of leaks in polymorphic containers. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 252–261, New York, NY, USA. ACM.

[23] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA. ACM.

[24] Hind, M., Burke, M., Carini, P., and Choi, J.-D. (1999). Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894.

[25] Joshi, P., Naik, M., Sen, K., and Gay, D. (2010). An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 327–336, New York, NY, USA. ACM.

[26] Klocwork. Klocwork k7. http://www.klocwork.com/.

[27] Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. (2005). Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12.

[28] Naik, M., Park, C.-S., Sen, K., and Gay, D. (2009). Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396, Washington, DC, USA. IEEE Computer Society.

[29] Neginhal, S. and Kothari, S. (2006). Event views and graph reductions for understanding system level c code. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 279–288.

[30] Quinn, C., Vilkomir, S., Parnas, D., and Kostic, S. (2006). Specification of software component requirements using the trace function method. In *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*, page 50.

[31] SciTools. Scitools understand. http://www.scitools.com/.

[32] Takahashi, J., Kojima, H., and Furukawa, Z. (2008). Coverage based testing for concurrent software. In *ICDCS Workshops*, pages 533–538. IEEE Computer Society.

[33] Taylor, R. N. (1983-04-01). Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84.

[34] Volanschi, N. (2006). A portable compiler-integrated approach to permanent checking. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 103–112.

[35] Xie, Y. and Aiken, A. (2005). Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 115–125, New York, NY, USA. ACM.

[36] Xie, Y., Chou, A., and Engler, D. R. (2003). Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC / SIGSOFT FSE*, pages 327–336. ACM.

[37] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. (2006). Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423.

[38] Yorsh, G., Ball, T., and Sagiv, M. (2006). Testing, abstraction, theorem proving: better together! In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 145–156.